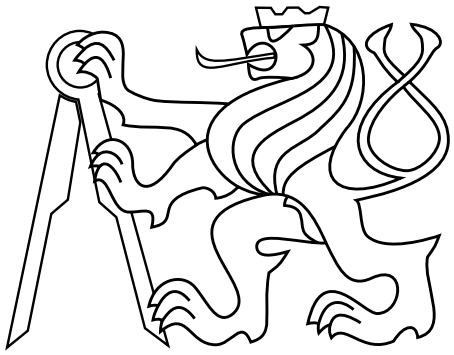
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics



Bachelor thesis

Tool for comparison of files of PLC programming and configuration software packages

Miroslav Nedvěd

Suprevisor: Ing. Pavel Vrba, Ph.D.

Study Program: Cybernetics and Robotics
Field of Study: Robotics

20.5.2014

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student: Miroslav N e d v ě d

Study programme: Cybernetics and Robotics

Specialisation: Robotics

Title of Bachelor Project: Tool for Comparison of Files of PLC Programming and Configuration Software Packages

Guidelines:

Goal of the work is to get familiar with ways of data representation used in the industrial automation systems (standard IEC 61131 for PLC programming) and to implement a tool that is able to compare two files and detect the changes (similarly to popular Windiff). Part of the work is the implementation of the graphical user interface, in which the differences will be presented to the user in a meaningful form. The work is done in collaboration with Rockwell Automation company, the world's leading automation solutions provider.
Content of work:

1. Get familiar with RSLogix5000 software package for programming and configuration of Rockwell Automation's Logix family of PLCs. Get also familiar with the structure of L5X file (XML format) that holds data of the RSLogix5000 project.
2. Create test data in form of two L5X files that will contain specific differences to be detected by the tool. Detection has to work for the following three major parts of the project: (i) tags in the global data table, (ii) tasks, programs, and routines including local data tables. The difference means addition or removal of given element, or modification of its parameters (like change of a data type of the tag). In case of routines the tools has to detect changes on the level of particular rungs and instructions.
3. Get familiar with existing ways of comparison of XML files and select the proper one, or design your own, method for effective comparison of given parts of L5X file.
4. Implement a Java library for comparison of two L5X files including the suitable format for holding the information about the detected changes.
5. Implement both textual and graphical user interface, in which the differences will be presented to the user in a vivid form.

Bibliography/Sources:

[1] Elhadi, M. ; Dept. of Comput. Sci., Sultan Qaboos Univ., Al-Khod, Oman ; Al-Tobi, A. Refinements of Longest Common Subsequence algorithm. 2010 IEEE/ACS International Conference on Computer Systems and Applications (AICCSA), 2010, pp. 1-5
[2] Extensible Markup Language (XML) 1.0 (Fifth Edition), dostupné z: http://www.w3.org/TR/2008/REC-xml-20081126/
[3] XML Path Language (XPath) 2.0 (Second Edition), dostupné z: http://www.w3.org/TR/xpath20/

Bachelor Project Supervisor: Ing. Pavel Vrba, Ph.D.

Valid until: the end of the summer semester of academic year 2014/2015

L.S.

doc. Dr. Ing. Jan Kybic
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 10, 2014

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student: Miroslav N e d v ě d

Studijní program: Kybernetika a robotika (bakalářský)

Obor: Robotika

Název tématu: Nástroj pro porovnání souborů SW nástrojů pro programování a konfiguraci PLC

Pokyny pro vypracování:

Cílem je seznámit se se způsoby reprezentace dat v průmyslových automatizačních systémech (standard IEC 61131 pro programování PLC) a vytvořit nástroj, který umožní porovnat dva soubory a detekovat jejich rozdíly (obdoba populárního nástroje WinDiff). Součástí řešení bude vytvoření prototypu uživatelského rozhraní, ve kterém bude možné rozdíly vizuálně prezentovat uživateli. Práce je vypracována ve spolupráci se společností Rockwell Automation, předním světovým hráčem v oblasti automatizačních řešení.
Náplň práce:

1. Seznamte se detailně s programem RSLogix5000 pro programování a konfiguraci PLC typu Logix firmy Rockwell Automation. Seznamte se dále se strukturou L5X souboru (formát XML) v němž je uložen projekt RSLogix 5000.
2. Vytvořte si testovací data v podobě dvou souborů L5X, které budou obsahovat specifické rozdíly, které bude vyvíjený nástroj detekovat. Detekce musí fungovat pro tyto tři hlavní části projektu: (i) tagy v globální datové tabulce, (ii) úlohy, programy a rutiny včetně lokálních datových tabulek, a (iii) add-on instrukce. Rozdíly je myšleno přidání nebo odebrání daného elementu, popř. modifikace jeho parametrů (např. změna datového typu tagu). V případě rutin musí nástroj jít s detekcí až na úroveň jednotlivých příček a instrukcí.
3. Seznamte se s existujícími způsoby porovnávání XML souborů a vyberte nejvhodnější z nich, popř. navrhněte vlastní, pro efektivní porovnání jednotlivých částí souboru L5X.
4. Naimplementujte knihovnu v jazyce Java pro porovnání dvou souborů L5X včetně vhodného formátu pro reprezentaci rozdílů.
5. Naimplementujte jak textové tak grafické uživatelské rozhraní, ve kterém budou uživateli rozdíly mezi porovnávanými soubory srozumitelně prezentovány.
6. Na testovacích datech demonstrujte funkci nástroje.

Seznam odborné literatury:

[1] Elhadi, M. ; Dept. of Comput. Sci., Sultan Qaboos Univ., Al-Khod, Oman ; Al-Tobi, A. Refinements of Longest Common Subsequence algorithm. 2010 IEEE/ACS International Conference on Computer Systems and Applications (AICCSA), 2010, pp. 1-5
[2] Extensible Markup Language (XML) 1.0 (Fifth Edition), dostupné z: http://www.w3.org/TR/2008/REC-xml-20081126/
[3] XML Path Language (XPath) 2.0 (Second Edition), dostupné z: http://www.w3.org/TR/xpath20/

Vedoucí bakalářské práce: Ing. Pavel Vrba, Ph.D.

Platnost zadání: do konce letního semestru 2014/2015

L.S.

doc. Dr. Ing. Jan Kybic
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 10. 1. 2014

Abstract

Goal of this work is to create a tool that will be able to compare two projects created in the development environment RSLogix5000 used for PLC programming. The project contains definitions of tags and programs written in ladder diagram of IEC61131 standard. The tool finds differences between two projects and presents them to the user.

First part of the presented tool compares similarity of an XML structure by using the information about the XML scheme. Second part of the tool compares textual representation of ladder diagrams. Lines without changes are matched as first by using method for finding the longest common substrings to find the same blocks of code. The unmatched lines are parsed into the object-structure containing list of elementary instructions to compute the similarity level. Last step is selecting the most similar common subsequence of instructions.

Algorithm used in the first part is much faster than commonly used algorithms for XML comparison, because the XML scheme is used and attributes *Name* are used as identifiers. Results of algorithm that is used in second part are very similar to differences that human can find and because unchanged rows were selected at first, it is very fast as well.

The proposed tool presents differences in a graphical form to the user. The results of comparison are presented in the same form as in RSLogix5000 and thus provide more useful information to the user than general text comparison tools. The proposed tool is also much more rich than a comparison tool RSLogix Compare provided in the standart RSLogix5000 instalation. The tool will be very usefull for programmers of industrial systems from Rockwell Automation company.

Abstrakt

Cílem této práce je vytvořit nástroj pro porovnávání dvou projektů vytvořených ve vývojovém prostředí RSLogix 5000 používaném pro programování PLC. Projekt se skládá z definice tagů a z programů zapsaných pomocí jazyka kontaktních schémat (standard IEC61131). Nástroj najde rozdíly mezi projekty a prezentuje je uživateli.

První část nástroje hledá rozdíly ve struktuře XML souborů s použitím informací které o struktuře známe. Druhá část se zabývá porovnáním textových reprezentací jazyka kontaktních schémat. Pomocí spojování nejdelších posloupností stejných řádků, se vyberou bloky nezměněného kódu a poté se zbylé řádky převedou do objektové struktury tvořené jednotlivými instrukcemi. Mezi těmito instrukcemi se vypočte procentuální podobnost a poté se hledá nejpodobnější společná posloupnost instrukcí.

Algoritmus první části je oproti běžným porovnávacím metodám XML souborů velice rychlý, protože používá známé informace o struktuře souborů a atribut *Name* jako identifikátor. Výsledky algoritmu druhé části odpovídají rozdílům, které by označil člověk a díky počátečnímu vybrání stejných bloků kódu, je algoritmus také velice rychlý.

Výsledný nástroj prezentuje uživateli informace o rozdílech v grafické podobě, která je podobná prostředí RSLogix 5000. Proto jsou informace pro uživatele mnohem užitečnější, než informace z běžně používaných nástrojů pro porovnávání textu. Výsledný nástroj je také mnohem bohatší než porovnávací nástroj RSLogix Compare, který je standartně dodávaný spolu s RSLogix 5000. Nástroj bude velmi užitečný pro programátory průmyslových systémů od firmy Rockwell Automation.

Acknowledgements

I would like to thank my supervisor, Ing Pavel Vrba, PhD for his help during work on this thesis and also my family for their support during my study.

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze dne ....

Podpis autora práce

Contents

1 Introduction

1.1 Context

Work is carried out on a request from Rockwell Automation company, which is among the top automation solutions providers. The flagship product is a family of control logic PLCs delivered together with programming tool RSLogix 5000.

During the development of projects for PLC regular backups are carried out. There is no solution how to compare these backups quckly and find all the differences between two versions of project instantly. This is necessary in case where the final version of the project does not work or the run of the project is several times slower.

There are version systems like SVN [1], CVS [2], Git [3] or other tools that can compare source code of programs. However, programming language of compared programs should be one of the commonly used (for example Java, C, Pascal, etc.) and the most importantly, compared languages should be textual. That is a problem because Ladder Diagram (see 2.3 on the following page) uses graphical diagrams for source code creation. Although Ladder Diagrams have their textual representation. This textual representation looks completely different than source codes written in commonly used languages.

There are tools such as WinDiff [4] used for comparing only textual files that can find different lines. When two lines are not completely identical, user has to compare them by himself. Therefore, these tools are inappropriate.

In addition, PLC project can be saved as XML formatted file. However, pure XML elements and attributes in this file are used only for storage information about project structure and definition of variables. Lines of source code representing lines of Ladder Diagrams are saved as text. That is why we get only differences between variables and project structure by using tools for comparing XML formatted files.

Finally, none of the existing tools can show differences in graphical form, like in Ladder Diagrams (see picture 1).

1.2 Goals

The aim is to create a tool that can compare two projects created in RSLogix 5000 saved as a file with the extension .L5X (page13) and to find the differences between each other.

- The tool will be designed for comparing files .L5X and their specific formatting. It may not be applicable to two general files.

- The tool will record the specific differences, determining the percent identity between the files is irrelevant.

- The tool will be able to show differences between rungs in ladder diagrams.

2 PLC programming

2.1 PLC

Programmable Logic Controller (PLC) was originally designed to replace relay circuits that have been used to automate in industry.

Therefore PLC is special computer designed to use in industry that performs the recorded program for process control in real time. PLCs have usually lesser failure rate than common computers and they are designed for multiple outputs and inputs. There are specific I/O cards which PLC interacts with sencors and actuators through in the controlled system.

2.2 IEC 61131

IEC 61131 is an International Electrotechnical Commission standard for programmable controllers. It is a set of requirements for advanced control systems. It is independent on the specific organization or company and has a broad international support.

IEC 61131 consists of the following parts:

Part 1: General information
Part 2: Equipment requirements and tests
Part 3: Programming languages
Part 4: User guidelines
Part 5: Communications
Part 6 is reserved
Part 7: Fuzzy-control programming
Part 8: Guidelines for the application and implementation of programming languages for programmable controllers

IEC 61131-3

IEC61131-3 specification is a part of IEC61131 (see 2.2) and deals with programming languages. Specification defines five different programming languages [5].

- Instruction List
- Structured Text Language
- Ladder Diagram
- Sequential Function Chart
- Function Block Diagram

Because of the fact that ladder diagram (LD) is the most commonly used programming language for PLC in the USA and development of this tool is assigned by Rockwell Automation company that comes from the USA, the tool will be focused on ladder diagrams.

2.3 Ladder Diagram

Ladder Diagram (LD) comes from relay wiring diagrams used by electricians (LD can be seen in Figure 2 on page 12). It was necessary to find simple way, how to teach electricians to programm instantly. Electricians are able to make same thing using LD. There is only one difference, they are not using paper, but computer.

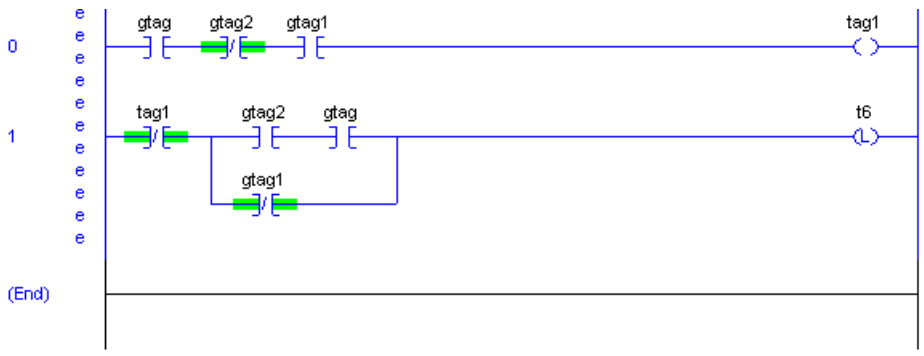


Figure 1: Example of ladder diagram

Program execution goes from top rung to bottom and from the left to the right on the rungs. Between instructions in the series logical AND is applied and between branches logical OR is applied. If program execution passes through all input instructions, an output instruction will make its action.

Every instruction in ladder diagram has its text representation. For example, text representation of rung number 0 in Figure 1 looks like this:

```
XIC(gtag)XIO(gtag2)XIC(gtag1)OTE(tag1)
```

This rung says that if tag named "gtag" is TRUE, tag "gtag2" is FALSE and tag "gtag1" is TRUE, program execution will pass through all input instructions and then the output instruction OTE will energize tag "tag1". However, if just one evaluation of instructions (XIC, XIO, XIC) returns FALSE, program execution will continue to the next rung.

Instructions

Instructions say to controller what to do with selected tags.

Simple instructions replace the function of relay. For example, XIC instruction (Examine If Closed) compares whether selected Tag is set. If so, program continues to the next instruction on the rung. If not, program continues to next the rung. There are also more complex instructions. For example, mathematical instruction ADD, that sums the values of the specified tags.

Instructions can by divided into two main categories.[6]

1. Input instructions - instructions that compare, check or examine - when they are evaluated as true, the program execution continues with the next instruction on the current rung.

2. Output instructions - instructions that take some action - they set value of the selected tag, turn off a device, turn on a device or copy data. For example ADD or OTE (Output Energize).

2.4 RSLogix 5000

RSLogix 5000 is development environment of the Rockwell Automation company. Created projects meet the IEC 61131 standard for programmable controllers (see 2.2 on the previous page).

Unlike other development environments such as NetBeans IDE, RSLogix 5000 is focused on creating programs only for PLCs. Projects can be created by using graphical environment without writting code.

2.4.1 Project structure

The structure can be seen in Figure 2 on the following page. It looks like the following:

Controller - Here you can define global variables that will be accessible from all programs in a project. Variables are called Tags.

Tasks - One project can have multiple Tasks, each of which may consist of multiple programs. Maximal number of tasks in one project is 16.

Program - Programs consist of several routines. In the program you can define local Tags, that will be accessible only in this program.

Routine - This is a place for logic of program.

Add-On instruction - They are similar to classes in object oriented programming languages with some limitations.

2.4.2 Tags

Variables in the environment RSLogix 5000 are called Tags. Each tag has defined name and type. Tags can be created in Controller Tags menu (global tags) or in Program Tags or AddOn-Instruction Tags menu (local tags).

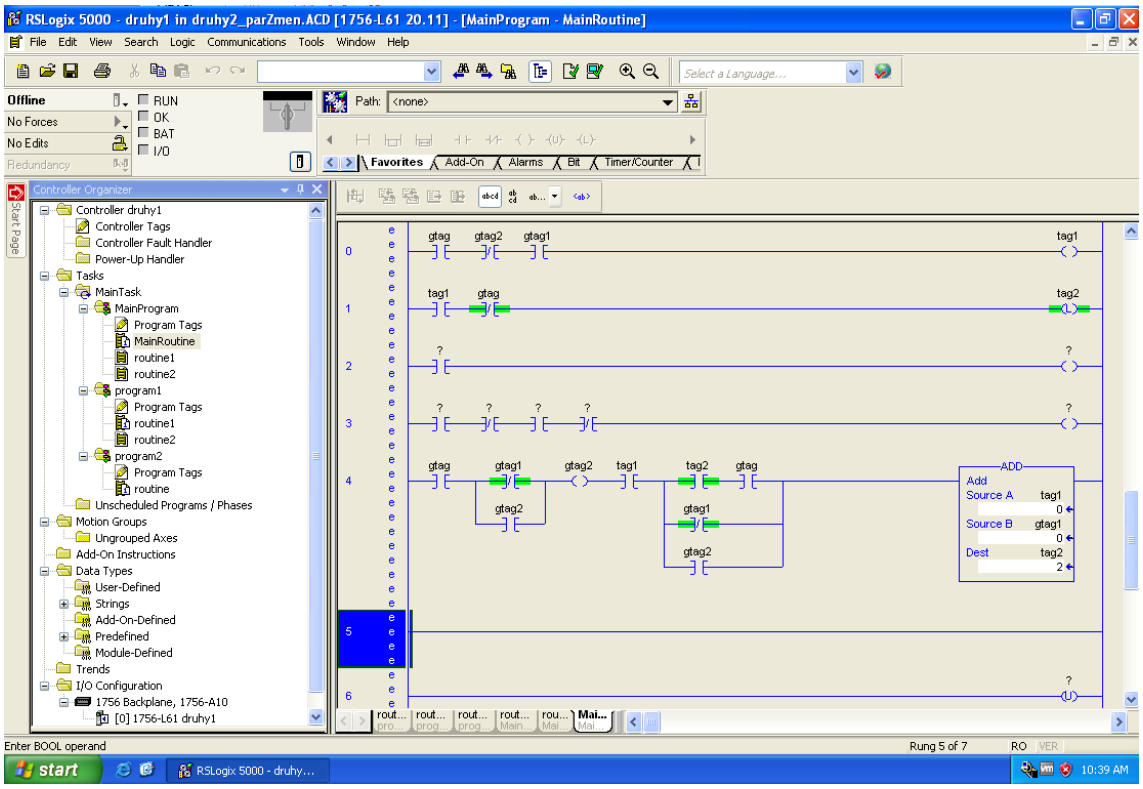


Figure 2: Preview of development environment RSLogix 5000

Data types can be simple like integer, boolean, string, multidimensional arrays etc.. We can define new data structures as well.

Naming rules by IEC61131-3 & RSLogix 5000 [5] say:

Fact 1. *Names for variables consist of a minimum of 1 character starting with an underscore "_" or an alpha character (A-Z or a-z), followed by one or more characters consisting of an underscore "_", alpha character (A-Z or a-z) or a number (0-9).*

2.4.3 Programs

Program consists of local tags and routines.

Local tags are accesible only from these routines. Routine from another program can not use local tags defined in this program.

Logic of program is stored in routines. If we are using LD (2.3) each routine contains rungs and one rung represents one line of source code. Routines can be performed periodically or on the bases of special events that can be, for example, modification of variable value.

2.4.4 Add-On instructions

Add-On instructions (AI) are similar to classes from object oriented programing languages, but for example, there is no possibility to use inheritance. It is appropriate to use Add-On instructions to replace commonly-used rows of logic and it is common that control engineers create libraries of AI that can be reused in multiple solutions.

We can define input and output parameters and local tags that are available only inside the Add-On instruction.

Source code of Add-On instructions is written in the same way as it is in routines, but code inside an Add-On instruction uses only the parameters and local tags defined in the instruction definition.

Saved Add-On instruction creates its own instruction block that can be inserted into selected rung simply as other common instructions.

Fact 2. *The name can be up to 40 characters long. It must start with a letter or underscore and must contain only letters, numbers, or underscores. The name must not match the name of a built-in instruction or an existing Add-On Instruction.[7]*

2.4.5 Structure of .L5X files

Project created in development environment RSLogix 5000 can be saved as text file with .L5X extension, default format is .ACD in binary proprietary encoding. Important thing is that these .L5X files have XML (see 3.1.1 on the next page) format and additionally basic structure of these files is same in all files generated by RSLogix 5000. Basic structure is shown in Figure 3.

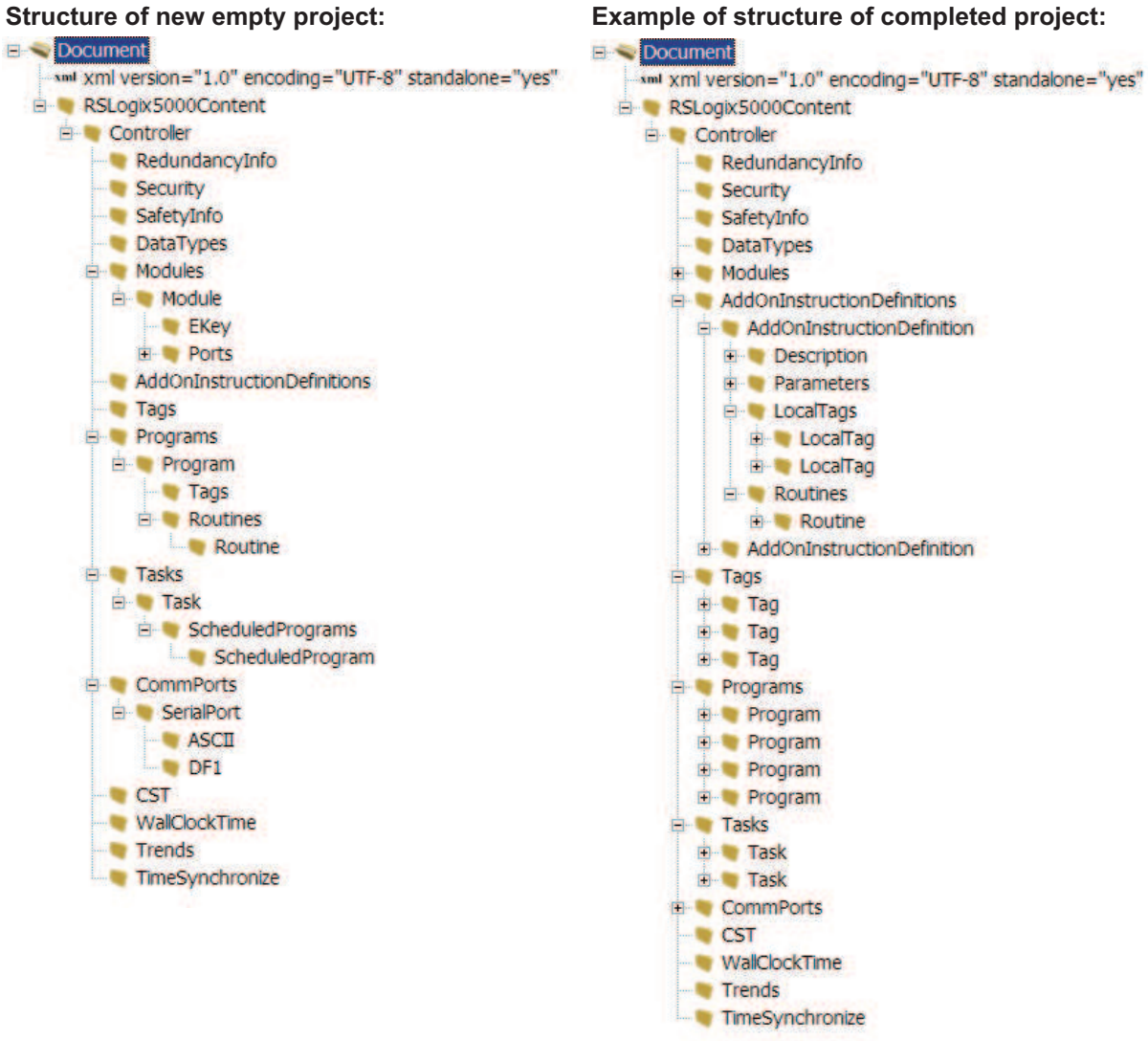


Figure 3: Structure of .L5X files

This basic structure says where the information about different parts of project is stored. For example, if we want an information about a program, we will look for it in the element Programs in the project structure.

Most of information about project are stored by using only XML elements and attributes. However, source code representating single rungs is saved as text in single element for each rung. So we can simply get text representation of logic of whole rung by using an XML parser, but we can not get code of the selected instruction by using an XML parser only. In case of rungs, we have to parse stored text and find instructions or branches by using some string parser.

Fact 3. *Big advantage of .L5X files over other common XML files is that many elements in .L5X file have attribute "Name". Every tag has name, every program has name, tasks, routines and Add-On instructions have names as well.*

2.5 RSLogix Compare

RSLogix Compare is a tool that is standardly provided with RSLogix 5000. It can recognize differences between tags and other parts of projects, but recognized differences in ladder diagrams are unsufficient. It can recognize changed tag if the rest of rung is identical, but neither this is done everytime. If there is one added instruction in the rung, rung without this instruction is recognized as deleted in the newer file and rung with this instruction is recognized as added into the newer file. The result is confusing and the user can not clearly see what is different.

You can see the result of RSLogix Compare used on simple routine in Figure 23 on page 43 and the result of LogixDiff used on same routine in Figure 24.

3 General algorithms for comparison

As it was said before (2.4.5), we can not use an XML parser only, because reasons would not be sufficient. Some parts of project are stored as a plain text, but in this plain text we can find logic of one line of algorithm. That is why we must select right combination of comparison methods to get the best reasons.

3.1 XML comparison

3.1.1 XML

"Extensible Markup Language (XML) describes a class of data objects called XML documents. XML is an application profile of Standard Generalized Markup Language [ISO 8879]." [8]

XML documents consist of elements that create tree-structure. Elements can contain either parsed or unparsed data. Some of parsed data form markup and some are saved as attributes of elements. Unparsed data are saved as an text in elements. The markup encodes a description of the document's storage layout and logical structure. [8]

The result is a format that is easy-readable for humans and computers.

3.1.2 XML processors

It is necessary to select the most suitable tool for browsing XML files. This tool will be an important part of LogixDiff, because we have to read from the files all the time. The time that takes reading of XML files can take major part in total running time of the LogixDiff.

Options:

Simple API for XML (SAX)

It is suitable for extensive documents. A document is read serially, so it is fast, but is difficult to use SAX for extracting information from the random places of XML document. [9]

Document Object model (DOM)

It creates a tree structure of the document where nodes represent elements. The entire document must be loaded into memory and that is memory intensive. [10]

Streaming API for XML (STAX)

It uses a cursor as the entry point. The application moves the cursor for getting the information. It is necessary to keep the track of location within the document. It is compromise between DOM and SAX. [11]

XML Path Language (XPath)

XPath is an expression language that allows processing of elements from an XML document. It is based on a tree structure of XML documents. It can use hierarchic addressing of the nodes in the XML tree. [12]

Java-based "document object model" (JDOM)

It is a Java representation of an XML document and it is optimalized for Java programmer [13]. For parsing documents it defaultly uses JAXP (Java API for XML Processing) [14] that leverages the parser standards SAX and DOM. Additionally JDOM is available under an Apache-style open source license.

JDOM java representation of an XML document:

An XML file is represented by tree structure, where nodes are called **Elements**. Each element can have **Attributes** that specify its properties. The elements can contain **Text** and other children elements.

Example:

```
<Tags>

    <Tag Name="t1" TagType="Base" DataType="DINT" Constant="false">
        <Data>01 00 00 00</Data>
        <Data Format="Decorated">
            <DataValue DataType="DINT" Radix="Decimal" Value="1" />
        </Data>
    </Tag>
    <Tag Name="t3" TagType="Base" DataType="DCI_STOP" Constant="false">
    ....

</Tags>
```

When we use JDOM to this XML structure we get element Tags with no attributes or text, but with child elements. The first child element is Tag that has attributes Name, TagType, DataType, Constant and it has child elements Data and Data. The first child element Data has no attributes but it contains text "01 00 00 00".

3.1.3 Methods

It is often faster and better to see changes between files than their current versions. For that purpose there are programs like diff, WinMerge and other, but algorithms of these programs work with text as a sequence of lines. We can use these algorithms to find differences between XML formatted documents, but they still handle it like sequence of lines.

Algorithms that work with XML documents like with tree-structured documents can gather more information about differences. These change detection algorithms are optimalized for different

Original structure:

```
<table>
     <tr id="1">
          <td>First line</td>
     </tr>
     <tr id="2">
          <td>Second line</td>
     </tr>
</table>
```

Updated structure:

```
<table>
     <tr id="2">
          <td>Second line</td>
     </tr>
     <tr id="1">
          <td>First line</td>
     </tr>
</table>
```

Figure 4: Example of the tree structure

requirements on time, memory or accuracy of results. Most of algorithms used for common XML files are looking for similarities in the tree-structure. This tree-structure can be ordered or unordered and this fact divides algorithms into first two groups.

When we have structure as in Figure (4), algorithms working with unordered tree do not find any difference. However, ordering among siblings is significant for algorithms that work with ordered trees. These algorithms mark the differences.

What kind of difference algorithm can find is another division of commonly used algorithms. Each algorithm can recognize added and deleted node and most of them can recognize updates. Last group of algorithms can recognize moves and copy. When we return to the Figure (4) and we use an algorithm for ordered trees, the algorithm will find one move if it can recognize moves. If not, it recognizes one deleted part and one added part. Recognizing of moves over the entire document is demanding, but the result with moves is smaller and more easily readable for the user. Results are saved in delta files. We should be able to create updated file by using the delta file and the original XML file .

Examples of tools for XML comparison are as follows:

X-Diff

The algorithm works with unordered trees and it can recognize added, deleted and updated elements [15]. It integrates an XML tree-structure with standard tree-to-tree correction technique [16], to find differences and to generate minimum-cost delta file.

XyDiff

XyDiff works with ordered trees and recognizes added, deleted, updated and moved elements [15]. It is focused on improving time and memory management, but it can not guarantee optimal results.

The algorithm computes hash and weight for every node in both XML trees and each node in original XML document gets its unique identifier. The algorithm matches nodes with the same value of hash and it tries to make the tree as large as possible.

DeltaXML

This tool can work with both, ordered and unordered trees and it recognizes added, deleted and updated elements [15]. Disadvantages of this tool are that it is commercial and maximum size of tree is 50 MB. On the other hand, it uses algorithm that runs in linear time.

Schema-Less, Semantic-Based Change Detection for XML Documents

As the title said, this method is not using an XML structure as the main source of information. This is an advantage in cases where structural changes of XML are significant, because algorithms

described above will break down in these cases. These structural changes happen particularly in data on the web.

The example of XML structures that can be compared by this method is shown in Figure 5. For more details about this method see [17]. For us this is just an example of method that is not using tree-to-tree comparison. We can not use advantages of this method, because we know the basic structure of files that will be compared (see 2.4.5) and this structure is same in all cases.

Original version:

```
<user>
   <nick>Joy</nick>
   <game>
      <name>Train</name>
      <score>550</score>
   </game>
</user>
<user>
   <nick>Pette</nick>
   <game>
      <name>Train</name>
      <score>280</score>
   </game>
</user>
```

Updated version:

```
<game>
   <name>Train</name>
   <score>550
      <user>
         <nick>Joy</nick>
      </user>
   </score>
   <score>280
      <user>
         <nick>Pette</nick>
      </user>
   </score>
</game>
```

Figure 5: Example of the same information considered in different XML schema

3.2 Text comparison

These methods and algorithms do not take meaning of a text into account. They work with the text as with sequence of characters. Most of tools such as Windiff [4] compare only whole lines of a text file. This should be much faster than to compare every single character and the results are very similar for the user.

Methods will be explained on one line of the text. However, you can imagine that one character in the line represents the whole row of the text.

Examples of text comparison algorithms:

Longest Common Subsequence

The longest common subsequence (LCS) problem is to find the longest sequence of characters that is the same in all compared strings [18]. Algorithms for computing LCS [19, 20] are used, for example, in file comparison tools like WinDiff [4], unix diff [21] etc.. The example of LCS you can see in Figure 7.

Longest Common Substring

The longest common substring (LCSS) problem is to find the longest string that is the same in all compared strings [22]. The example of LCSS you can see in Figure 7.

Levenshtein distance

It says how dissimilar two strings are [23]. It counts minimal amount of operations (insertion, deletion, substitution) that are needed to make strings identical. This method can be used, for example, for dictionaries or some whisperers. It is an easy way to find similar word or to say how much two words are different. However, it is not the best way for comparison of extensive text files.

For example, if we compute Levenshtein distance between word "head" and "hands", the result will be number three (see Fig. 6).

```
head
haad    - substiotution for "e" to "a"
hand    - substiotution for "a" to "n"
hands   - insertion of "s"
hands
```

Figure 6: Example of computing Levenshtein distance

Differences between LCS and LCSS methods

In Figure 7 you can see example of LCS and LCSS methods. We prepare two strings and compare them with both methods. At first you can see the sequence of characters selected by using LCS and LCSS methods, which were used only once. These sequences are the longest selected ones. Secondly you can see all subsequences selected by using methods recursively on unmatched characters until their results are not empty.

At the end of the Figure you can see how the characters are matched. Notice that it is the same in both methods.

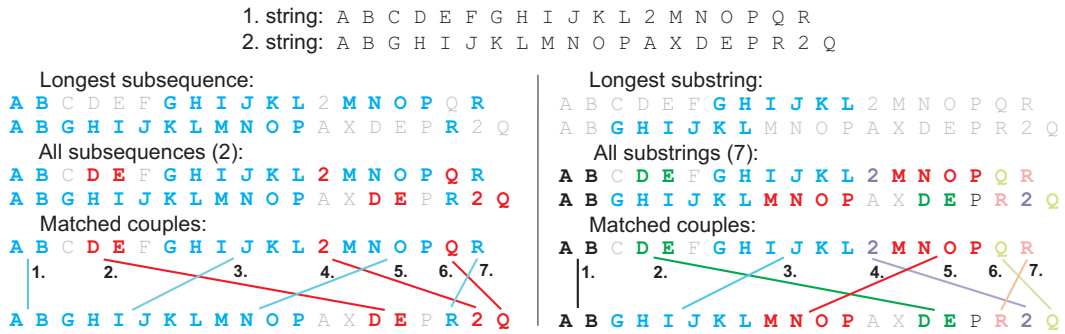


Figure 7: First example of differences between LCS and LCSS results

Only in special cases results of recursively used methods are dfferent. For example, where some characters are copied after substring moved from the end of first string to the beggining of second string and this moved substring is shorter than the rest of strings and additionally in the rest of the string there are some little changes. These little changes break the LCSS method on the beggining of the first string and it matches moved substring, but LCS method skips little changes and matches the first part of the string (see Fig. 8).

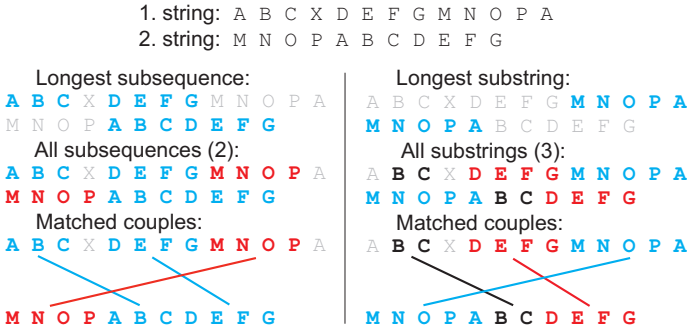


Figure 8: Second example of differences between LCS and LCSS results

3.3 Source code comparison

The typical thing is that these methods parse text file to some atomic elements (tokens) at first. To recognize what is atomic element and what is its function, tables of known tokens are used. These are different for different languages. The comparison itself works with these tokens. Some methods use text comparison methods modified for working with tokens instead of characters (Levenshtein Distance, LCS, see 3.2). Some methods make a tree-structure from tokens and compare similarities between tree structures. They either use a combination of tree structure and text comparison methods.

Source code comparison methods in this article [24] are divided into Textual comparison, Token Comparison, Metric comparison, Comparison of abstract syntax trees and Others.

Examples of known algorithms:

The NCP Algorithm of Fuzzy Source Code Comparison

This algorithm works with tokens carried in trie-trees. It uses Levenshtein distance between tokens for similarity computation. The algorithm is used for recognizing plagiarism. [25]

Clone Detection Using Abstract Syntax Trees

Clone detection tries to find parts of codes that compute same results [26]. These parts are mostly produced by copy-paste operation. Therefore semantic equivalence of programs is not detected completely. For the same reason, commentaries, spacing or other non-semnantic changes are expected not to be modified.

The first step of algorithm is creation of abstract syntax tree. Then sub-trees are detected. In the next step sequence of sub-tree clones is found. In the last step clones are tried to be found by combinations generalizing of other clones.

4 Proposed solution

At the beggining of developing LogixDiff we have to select some tool to handle information stored in XML documents. We have selected JDOM browser (see 3.1.2). There are several reasons for this decision, like:

1. LogixDiff have to be created in Java language and JDOM is optimalized for Java.
2. The most usefull thing is that elements from an XML document are represented as Java objects and they implement functions to work with them.
3. These given objects are still in tree structure.
4. There is a possibility to change used parser or to use XPath language and that is good to know to start the project, which basis will stay on this tool.

XML comparison method

We can say that our algorithm works with unordered trees, because order between siblings is not significant (see 3.1.3). However, we do not compare tree-structures of XML documents, because we know that the basic structure of used files is the same (see 2.4.5). Information about used project structure giving us a possibility to optimalize the algorithm to be faster, with no impact on the accuracy of the obtained results.

Another fact arising from the known structure is that we do not have to look for moved or copied elements, because for example element Tag can not be moved between Programs etc.. So we recognize only added, deleted and changed elements and file with results is still minimum-sized.

These facts are used in both proposed methods, Tags comparison and Logic comparison. Each comparison of whole L5X document uses both methods together to reach the best results.

4.1 Tags comparison

Tags comparison method uses facts that we know about common XML files and additionally specific properties of L5X files (see 2.4.5 on page 13). It can find all differences in projects but differences found between rungs are insufficient, because this method can only detect if rungs are completely identical or not.

The method is used for finding differences between properties of programs, structure of projects, tags and all other infromation stored in L5X files. It is not used for finding differences in routines of programs and add-on instructions, that means differences between single rungs.

Basic preconditions:

1. Proposed algorithm uses JDOM for reading files, so files have to be XML formatted. It would be inefficient not to use advantages of XML formatting.
2. Algorithm compares only the same elements which means that for example element Tags from first file is compared only with element Tags from second file. It does not make sense to compare element DataValue with element Program, because we know that different elements carry information about different part of the project (see 2.4.5 on page 13).
3. It uses fact that lots of elements have attribute Name (Fact 3 on page 14). If an element has attribute Name, the element from the second file must have this attribute as well and the attribute must have same values.
4. We know that in one file can not be two programs with the same name, two global tags with the same name and in one program can not be two routines with the same name. In general we can say, that in the same deep in project structure and under the same parent element can not be two same elements with identical values of attribute Name. So if we find two elements with the same name and with identical values of attribute Name with belonging to the same parent element (but in another file) these elements are original one and its updated copy.
5. Basic project structure of .L5X files is the same (see 2.4.5 on page 13). We know that if wanted element is under one parent element in one file this wanted element must be under the same parent element in second file. If wanted element is not there, it is deleted. So we can simply recognize added or deleted elements and we do not have to look into this elements and look at their structure.

You can see that defaultly there are not compared elements with different name or with different value of attribute "Name". However, user can specify paths to wanted parts of L5X files and then these parts will be compared.

4.1.1 Algorithm description

At the beggining we have two L5X files Older and Newer. Older file represents original file and Newer file represents updated file.

Algorithm step 1 - The algorithm connects to L5X files and loads root elements rootOlder and rootNewer and then objects (pathOlder and pathNewer) representing path to these root elements in XML files are created (see algorithm 1 on the following page in pseudocode).

Algorithm step 2 - These roots and paths are passed as arguments to function compareByXML and it recursively checks all elements in L5X file and returns all differences found. If the passed older element is Programs or AddonInstructionDefiniton the function returns empty diffs without comparison, because programs and add-on instructions are compared by second proposed method (see chapter 4.2 on page 24).

Algorithm step 2 a) - Attributes of elements are selected and empty map of their couples is prepared (see algorithm 2 on the following page in pseudocode).

Algorithm step 2 b) - Starts a loop to compare each attribute of attributesOlder with each attribute of attributesNewer:

Algorithm 1 main method of Tags comparison algorithm (pseudocode)

```
1.
roodOlder = loadRootElement(olderFile)
roodNewer = loadRootElement(newerFile)
pathOlder = rootOlder.getPath();
pathNewer = rootNewer.getPath();
2.
compareByXML(rootOlder, rootNewer, pathOlder, pathNewer)
```

If attribute has no couple in the map, checks if the names of attributes are identical, if they are, saves this couple to the map and checks if values of attributes are same, if values are different adds all information about paths, names and differences to Diffs.

Algorithm step 2 c) - Looks for attributes with no couple in the attributeMap and adds them to diffs

Algorithm step 2 d) - Checks difference between text of elements.

Algorithm step 2 e) - Creates lists of children of given elements and makes map of couples childrenElementsMap (see algorithm 3 on the next page in pseudocode). If element has couple in childrenElementsMap function compareByXML is recursively used to them and its differences are added to list of differences. If element has no couple in childrenElementsMap information about it is added into the list of differences.

Algorithm 2 description of function compareByXML (pseudocode)

```
compareByXML(olderElement, newerElement, pathOlder, pathNewer)
    diffs = new empty list of differences
   IF olderElement IS NOT Programs AND olderElement IS NOT AddOnInstructionDefinition
2 a)
      attributesOlder = olderElement.getAttributes()
      attributesNewer = newerElement.getAttributes()
      attributeMap = array[2]
2 b)
      FOR i < number of older attributes
         FOR j <  number of newer attributes
            IF there is no matched couple in attributeMap[1][j]
               IF names of attributeOlder is equal to name of attributeNewer
                  adds both to attributeMap
                  IF values of attributes are not identical
                     adds new diff to diffs
2 c)
      FOR i < number of older attributes
         IF attributeMap[0][i] == -1
            adds new diff to diffs
      FOR i < number of newer attributes
         IF attributeMap[1][i] == -1
            adds new diff to diffs

2 d)
      IF text of elementOlder is not equal to text of elementNewer
         adds new diff to diffs
2 e)
      childrenElementsMap = makeChildrenElementsMap(elementOlder, elementNewer)
      FOR i < number of olderElement children
         IF child has couple in childrenElementsMap
            adds result of compareByXML(childOlder, childNewer, updated pathOlder, updated
                pathNewer) to diffs
         ELSE
            adds new diff to diffs
   RETURN diffs
```

Algorithm 3 description of function makeElementsMap (pseudocode)

```
makeChildrenElementsMap(elementOlder, elementNewer)
   elementsMap = array[2] where value on the selected position means matched element from
       second file
   FOR i < number of older elements
      FOR j < number of newer elements
         IF actual newer element has no couple in elementsMap
            IF name of elementOlder is equals to name of elementNewer
               IF attribute "Name" is null in both elements
                  saves this couple to the elementsMap
               ELSE IF attribute "Name" is NOT null in both elements
                  IF value of attribute "Name" is identical in both elements
                     saves this couple to the elementsMap
   RETURN elementsMap
```

4.1.2 Expressing differences

Each difference carries information about positions in both L5X files that means about its positions in projects structure, information about what type of difference it is (added, deleted, changed) and information about what type of XML part is changed (element, text, attribute).

Recognized differences

Three types of differences are defined:

1. added - Different part was added to newer file. This element is not in the older file.
2. deleted - Different part was deleted from newer file. This part is in the older file, but it is not in the newer file.
3. changed - Different part was changed in newer file. This part is in both files, but it has different values.

Each type of difference can be in Attribute, Element can be added or deleted and Text can be only changed.

Position in project structure

Position in project structure is saved as path consisting of three parts. Every part carries information about way from the root element through parent elements to this modified element.

First part is a list of strings, where each string is name of element.

Second part is a list of numbers, where each number means order between children under actual parent element.

Third part is a list of strings, where each string is value of actual attribute Name.

List of numbers is only one of these that has right unique path in every situation, because each element does not have attribute Name and there are some situation where are more than one element with same name, but with no attribute Name (see 4.1.2).

Lists of strings are mainly for users, because if we show to the user path of numbers it will be difficult for him to find wanted element. However, we can get strings by using list of numbers, but it is faster to save them than to find it by using the path of numbers. We assume that there will not be too differences to fill memory, but computing time can be long.

Example:

Example of differences is shown in Figure 9. We can see four differences there.

1. element Tag with name "gtag" is deleted.

Path in older file: Controller(example)/Tags/Tag(gtag)

2. text of element Data was changed.

   Path: Controller(example)/Tags/Tag(X)/Data
   Older value: 0
   Newer value: 16

3. value of attribute of element DataValue was changed.

   Path shown to user: Controller(example)/Tags/Tag(X)/Data/DataValue
   All tree parts of path:

   a) elements: Controller/Tags/Tag/Data/DataValue
   b) order in older file: 0/5/1/1/0
   c) names: example/-/X/-/-

   Ordered path in newer file is different: 0/5/0/1/0
   Attribute: DataType
   Older value: DINT
   Newer value: REAL

4. element Tag with name "check" is added.

   Path in newer file: Controller(example)/Tags/Tag(check)

Older:

```
<Controller Use="Target" Name="example">
   <RedundancyInfo Enabled="false" />
   <Security Code="0"/>
   <SafetyInfo/>
   <DataTypes/>
   <AddOnInstructionDefinitions/>
   <Tags>
      <Tag Name="gtag" TagType="Base">
         <Data></Data>
         <Data Format="String" Length="0">
            <![CDATA['']]>
         </Data>
      </Tag>
      <Tag Name="X" TagType="Base" >
         <Data>0</Data>
         <Data Format="Decorated">
            <DataValue DataType="DINT"/>
         </Data>
      </Tag>
   </Tags>
   ...
```

Newer:

```
<Controller Use="Target" Name="example">
   <RedundancyInfo Enabled="false" />
   <Security Code="0"/>
   <SafetyInfo/>
   <DataTypes/>
   <AddOnInstructionDefinitions/>
   <Tags>
      <Tag Name="X" TagType="Base" >
         <Data>16</Data>
         <Data Format="Decorated">
            <DataValue DataType="REAL"/>
         </Data>
      </Tag>
      <Tag Name="check" TagType="Base">
         <Data>0</Data>
         <Data Format="Decorated">
            <DataValue DataType="BOOL"/>
         </Data>
      </Tag>
   </Tags>
   ...
```

Figure 9: Example of difference types

4.1.3 Evaluation

Tags comparison method is maximally optimalized for our needs. We do not compare tree structures, but we use known information about the project structure (see 2.4.5). Algorithm finds all differences in project structure, global tags, tasks etc., it is not used for comparison of Programs and AddOnInstructions. Each difference carries information about position in the project structure. So users have complete information about what is changed in what part of their projects.

Algorithm complexity is O(mn) to compare two documents with number of elements **n** and **m**. However, number of executed operations is much smaller, because we compare only elements on

the same position in the project structure. Additionally if we do not match couple of elements we do not look on their children elements. On the other side if we match couple of elements, second loop stops.

Implementation of algorithm can be found on attached CD. There you can see used classes for storing differences and concrete source codes for comparison.

Example of operations executed for elements matching.

Files structures are in Figure 10. In older structure is 30 elements and in newer structure is 29 elements. Theoretically, we would need $n \cdot m = 30 \cdot 29 = 870$ operations.

Our rules:

I. Algorithms compares elements only on same position in project structure.

II. Our algorithm stops second loop when two elements are matched.

III. Matched elements are not compared again.

If we use these three simple rules, we need only 36 operations to compare these files. All executed compares are in Figure 10.

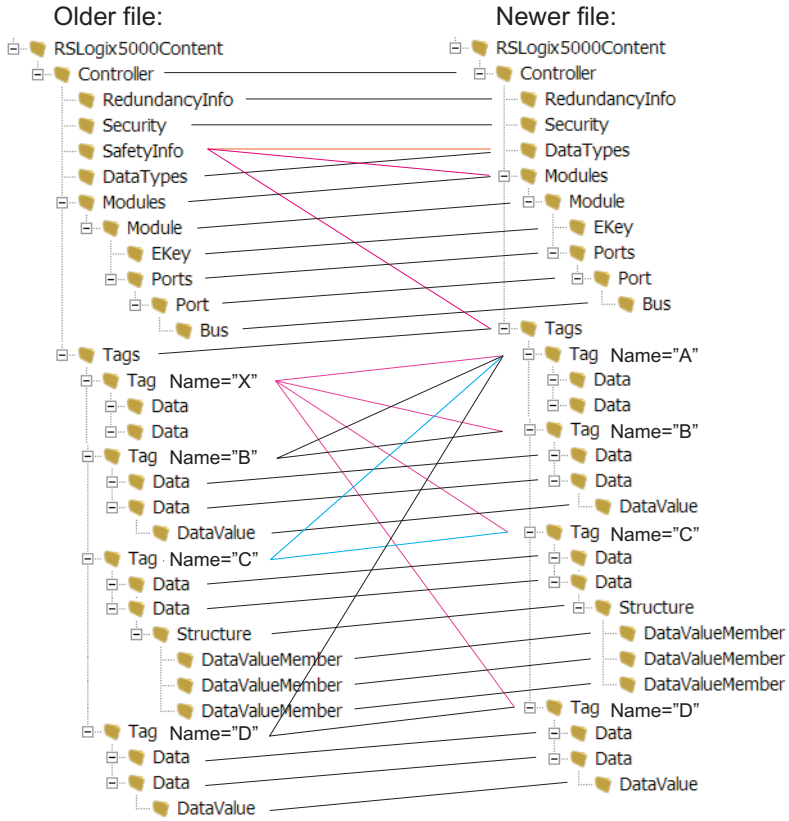


Figure 10: Example of executed comparisons

4.2 Programs comparison

Programs comparison uses both methods Tags comparison method and new method (Logic comparison method), which is described in this chapter.

Obtaining more accurate results while comparing logic of single rungs is main reason for the development of the Logic comparison method. As it is written in chapter 4.1, Tags comparison method can recognize only if logic of two rungs is completely identical or not.

Logic comparison method is proposed to compare rungs of programs. It can find differences between single instructions.

Programs comparison runs in five steps:

Tags comparison method:

1. Pairs programs by attribute "Name".
2. Loads minimal information about programs (see 4.2.1).

Only matched couples of programs are compared in next steps.

3. Finds differences between local tags and matched routines. It is not looking to children elements of single routines

Logic comparison method:

4. Selects rungs that are identical in original and updated routine and pairs these rungs by finding the longest common substrings.
5. Compares all unpaired rungs by finding the longest common subsequence between its instructions. Selects couples that have the biggest similarity and saves differences between them.

4.2.1 Minimal information about programs

Minimal information about programs is only visualization of programs structure for user. It is mainly used for comparisons operated from command line that have only textual output. In minimal information user can see whether some program was probably only renamed and based on that he can decide to compare this program and its renamed copy. Renamed programs are not compared, because only programs and routines with same name are compared defaultly.

Minimal information consist of:

- program name
- names of all routines in this program
- numbers of rungs in each routine

In Figure 11 there is example of minimal information about programs printed into command line. Notice that program "zero" is probably renamed copy of program "prog2".

```
      Older file:
program name: MainProgram
0. routine name: MainRoutine, rungs: 13
1. routine name: routinre01, rungs: 7

program name: prog2
0. routine name: routine20, rungs: 7
1. routine name: routine21, rungs: 3

program name: programX
0. routine name: routineX1, rungs: 5
```

```
      Newer file:
program name: MainProgram
0. routine name: MainRoutine, rungs: 13
1. routine name: routA, rungs: 0
2. routine name: routinre01, rungs: 7

program name: programX
0. routine name: routineX1, rungs: 5

program name: zero
0. routine name: routine20, rungs: 7
1. routine name: routine21, rungs: 3
```

Figure 11: Example of minimal information output

4.2.2 Matching identical rungs

This part tries to find all sequences of rungs that are unchanged or only moved. It can recognize added or deleted rungs only in case, that each rung of one routine has couple and second routine has more rungs. Results of this part correspond with changes that could be done by programmer.

This method can not recognize differences between instructions, but it is fast way to select modified and identical rungs.

Routine stored in L5X file is a combination of XML language and textual strings that represent ladder diagrams. Used XML structure is the same in all cases. So we can select only strings that represent ladder diagrams and we can work only with them in following steps. Example of routine stored in the L5X file is shown in Figure 12.

```
<Routine Name="MainRoutine" Type="RLL">
      <RLLContent>
            <Rung Number="0" Type="N">
                  <Text>
                        <![CDATA[XIC(A)OTE(B);]]>
                  </Text>
            </Rung>
            <Rung Number="1" Type="N">
                  <Text>
                        <![CDATA[XIC(A)XIC(B)XIC(?)XIC(?)ONS(X)OTL(A);]]>
                  </Text>
            </Rung>
            <Rung Number="2" Type="N">
                  <Text>
                        <![CDATA[XIO(i)XIC(A)OTE(b);]]>
                  </Text>
            </Rung>
      </RLLContent>
</Routine>
```

Figure 12: Example of routine stored in L5X file

Algorithm description

At the beggining we have two lists of Strings that represent all rungs in older and newer routine.

Comment for algorithm 4 on the next page:

1. Each rung from first list is compared with each rung from second list. If they are equal the algorithm saves number that says how many rungs in the sequence are same into similarityMatrix.

Rows in the similarityMatrix represent older rungs and columns represent newer rungs.

2. Starts a loop for getting all rungs that have pair.

2 a) The longest substring between unmatched values in the similarityMatrix is selected here (the biggest number in matrix).

2 b) Single rungs from selected substring are matched.

Two dimensional array carries information about couples.

2 c) Some rows of the matrix are reserved now. It is necessary to check columns that are one row below these reserved rows. If they have bigger value than one, it means, that the substring, which lenght is represented by this value, starts somewhere between or above the reserved rows. We have to change these values to 1 and update all next values of this substring. (see example on page 28).

2 d) The same process as 2 c) however, we check rows that are one column on the right from reserved columns.

Result of this function is two-dimensional array of integers (couplesMap[2][]), where all information about couples is stored. Array couplesMap[0] represents rungs from older file, position of x (couplesMap[0][x]) means rung from older file and value of couplesMap[0][x] means matched couple from newer file. Array couplesMap[1] represents rungs from newer file, position of y (couplesMap[1][y]) means rung from second file and value on this position means matched couple from

Algorithm 4 matching rungs by using longest common substring method (pseudocode)

```
1.
FOR i < number of rungs in older file
   FOR j < number of rungs in newer file
      IF rungsO[i] EQUALS rungsN[j]
         similarityMatrix[i][j] = 1 + similarityMatrix[i-1][j-1]
      ELSE
         similarityMatrix[i][j] = 0
2.
WHILE longestSubString IS NOT 0
2 a)
   longestSubString = 0
   coordinates = null
   FOR i < number of rungs in older file
      IF rungsO[i] IS NOT matched
         FOR j < number of rungs in newer file
            IF rungsN[j] IS NOT matched
               IF similarityMatrix[i][j] > longestSubString
                  longestSubString = similarityMatrix[i][j]
                  coordinates = [i,j]
2 b)
   FOR i < longestSubString
      couplesMap[Older][coordinates[row]-i] = coordinates[column]-i
      couplesMap[Newer][coordinates[column]-i] = coordinates[row]-i
2 c)
   i = coordinates[row] + 1
   FOR i < number of rows
      IF similarityMatrix[i][coordinates[column] + 1] > 1
         WHILE nextNumber > 1
            similarityMatrix[i + step][coordinates[column] + 1 + step] = 1 + step
            nextNumber = similarityMatrix[i + step + 1][coordinates[column] + 1 + step + 1]
            step ++
2 d)
   j = coordinates[column] + 1
   FOR j < number of columns
      IF similarityMatrix[coordinates[row] + 1][j] > 1
         WHILE nextNumber > 1
            similarityMatrix[coordinates[row] + 1 + step][j + step] = 1 + step
            nextNumber = similarityMatrix[coordinates[row] + 1 + step + 1][j + step + 1]
            step ++
```

older file. Notice, that arrays do not need to have same lenght. If value is -1, selected rung has no couple in second file.

We need information about all rungs and that is reason why we use two-dimensional array although the information about matched rungs is duplicated. We can get unmatched rungs very quickly from two-dimensional array.

Example

If we use described algorithm to these lists of strings:

rungsO: {a,a,c,d,e,f,g,f,g,h,d}
rungsN: {a,b,c,d,e,f,g,h,d,a,c,x}

Algorithm step 1 - We get similarity matrix a) which is shown in Figure 13.

Algorithm step 2 a) - First longest substring has lenght 5, it is substring {c,d,e,f,g}, it can be seen in matrix a) in Figure.

Algorithm step 2 b) - All used rows and columns are reserved, see matrix b) in Figure. Rungs from the longest substring are matched. Our array is now:

$$couplesMap[0]: \quad -1 \quad -1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad -1 \quad -1 \quad -1 \quad -1$$
$$couplesMap[1]: \quad -1 \quad -1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad -1 \quad -1 \quad -1 \quad -1 \quad -1$$

Algorithm step 2 d) - While checking rows, the algorithm finds value 3. This value is changed to 1 and all values that continue with this substring are updated. See differences between matrix b) and c).

a)

	a	b	**c**	**d**	**e**	**f**	**g**	h	d	a	c	x
a	1	0	0	0	0	0	0	0	0	1	0	0
a	1	0	0	0	0	0	0	0	0	1	0	0
c	0	0	**1**	0	0	0	0	0	0	0	2	0
d	0	0	0	**2**	0	0	0	0	1	0	0	0
e	0	0	0	0	**3**	0	0	0	0	0	0	0
f	0	0	0	0	0	**4**	0	0	0	0	0	0
g	0	0	0	0	0	0	**5**	0	0	0	0	0
f	0	0	0	0	0	1	0	0	0	0	0	0
g	0	0	0	0	0	0	2	0	0	0	0	0
h	0	0	0	0	0	0	0	3	0	0	0	0
d	0	0	0	1	0	0	0	0	4	0	0	0

b)

	a	b	c	d	e	f	g	**h**	**d**	a	c	x
a	1	0	-	-	-	-	-	0	0	1	0	0
a	1	0	-	-	-	-	-	0	0	1	0	0
c	-	-	*	-	-	-	-	-	-	-	-	-
d	-	-	-	*	-	-	-	-	-	-	-	-
e	-	-	-	-	*	-	-	-	-	-	-	-
f	-	-	-	-	-	*	-	-	-	-	-	-
g	-	-	-	-	-	-	*	-	-	-	-	-
f	0	0	-	-	-	-	-	0	0	0	0	0
g	0	0	-	-	-	-	-	0	0	0	0	0
h	0	0	-	-	-	-	-	**3**	0	0	0	0
d	0	0	-	-	-	-	-	0	**4**	0	0	0

c)

	a	b	c	d	e	f	g	h	d	a	c	x
a	1	-	-	-	-	-	-	-	-	-	-	-
a	-	-	-	-	-	-	-	-	-	1	-	-
c	-	-	1	-	-	-	-	-	-	-	-	-
d	-	-	-	2	-	-	-	-	-	-	-	-
e	-	-	-	-	3	-	-	-	-	-	-	-
f	-	-	-	-	-	4	-	-	-	-	-	-
g	-	-	-	-	-	-	5	-	-	-	-	-
f	-	0	-	-	-	-	-	-	-	-	0	0
g	-	0	-	-	-	-	-	-	-	-	0	0
h	-	-	-	-	-	-	-	1	-	-	-	-
d	-	-	-	-	-	-	-	-	2	-	-	-

Figure 13: Example of similarity matrix in some steps of running algorithm

Matrix c) shows all reserved rows and columns and all selected couples, when loop from algorithm step 2 ends.

Final array with couples:

$$couplesMap[0]: \quad 0 \quad 9 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad -1 \quad -1 \quad 7 \quad 8$$
$$couplesMap[1]: \quad 0 \quad -1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 9 \quad 10 \quad 1 \quad -1 \quad -1$$

4.2.3 Single rungs comparison

We want to find visual similarity between rungs, thus for example logical instructions that are swapped are a difference for us. Although swapped logical instructions change nothing in program functionality.

This part of programs comparison starts after method for identical rungs matching. So we can use the array of couples, which is result of previous part. Firstly, this method selects rungs that have no couple in array (values -1 on their position). Each of this rungs from older file is compared with each of rungs from newer file and similarities between all of them are computed. Couples are matched by similarities, the most similar rungs are matched as first. Rungs with similarity less than 30% are marked as deleted or added, because in these cases it can be assumed that rung is not the original and its updated copy. This value is selected empirically, because there is no possibility how to detect what programmer has really made and because rungs with this similarity are very different.

Parsing rung to single elements

Rung is parsed into list of elements, where element can be atomic or branching. Atomic element is an instruction and its tags that is represented by name of instruction and list of tags names. Branching consists of list of lists of elements.

By using this simple structure we can parse entire rung with branchings in branchings etc. and thus we can create structure of objects in Java. Example of parsed rung can be seen in Figure 14.

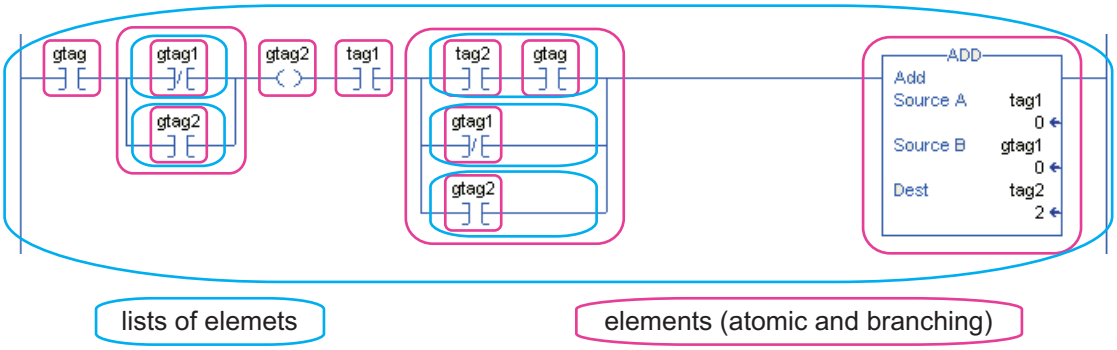


Figure 14: Example of parsed rung

Parsed rung comparison

At first, similarity between each element from older rung with each element of newer rung is computed. You can imagine that these similarities are stored in matrix where rows represent elements from older rung and columns represent elements from newer rung. We want to find visual differences so order between elements is important. That is a reason for finding the longest common subsequence (see 3.2 on page 17) between elements, because LCS finds the largest number of elements that have same order in both rungs.

Problem is, that values stored in similarity matrix are not only 1 and 0, but they occur in interval $<0;1>$. So one element can have better similarity than many others (that are not zeros) together. Thus it is not enough to find the largest number of elements, but we have to find the largest sum of similarities of selected sequence of elements. This can be done by finding the longest weighted way in ordered graph.

Computation of similarity:

Each computed similarity **S** occurs in interval $< 0; 1 >$.

1. Similarity between atomic instructions:

If two instructions have same name, their similarity is $S_I = S_I + 0.5$. Additionally similarity for two instructions with same number of tags tn is $S_I = S_I + \frac{0.5}{tn}$ for every identical tag that has same order in both instructions.

Similarity between two instructions is:

$$S_I = x \cdot 0.5 + \sum_1^a \frac{0.5}{tn}$$

Where if instructions name are not same $x = 0$ and if instructions have same name $x = 1$ and where instructions have number of same tags in same order a.

2. Similarity between branchings:

   Firstly similarities between all branches are computed. The most similar branches are matched as couples, their similarities are $bs_1 ... bs_n$ where n is number of branches in branching with less number of branches. Similarity between two branchings with number of branches bn_1 and bn_2 is:

   $$S_B = 2 \cdot \frac{\sum_{i=1}^{n} bs_i}{bn_1 + bn_2}$$

3. Similarity between atomic element and branching:

   This similarity is set to zero. $S_{IB} = 0$

4. Similarity between single branches:

   It is computed in same way as similarity between rungs.

5. Similarity between rungs:

   Firstly similarities between all elements on rung are computed. If similarity is bigger than 0.1 (we do not need zeros) it is saved as node of graph to list of nodes. Each node carries information about order in older and newer rung and about its similarity.

   You can imagine that similarities are stored in matrix (see Fig. 15 a)) where rows are elements in older rung and columns are elements in newer rung. We want to select couples that have the biggest sum of similarities and we do not want to match couples to get the crossing. So we need to solve the longest common subsequence problem by finding the most similar common subsequence. Selecting LCS in matrix means that if some couple C is matched on coordinates [r,c], next matched couple has to be minimally one line below and one column right from couple C, which means that the nearest allowed coordinates are [r+1,c+1]. In our graph finding LCS means, that there is a ordered way from each node N with coordinates [r,c] to each node X with coordinates [r+z,c+z] where $z > 0$ (see Fig. 15 c)). There should be many zero similarities in the matrix that is why we make graph from non-zero similarities only (see difference between Fig. 15 a) and 15 b)).

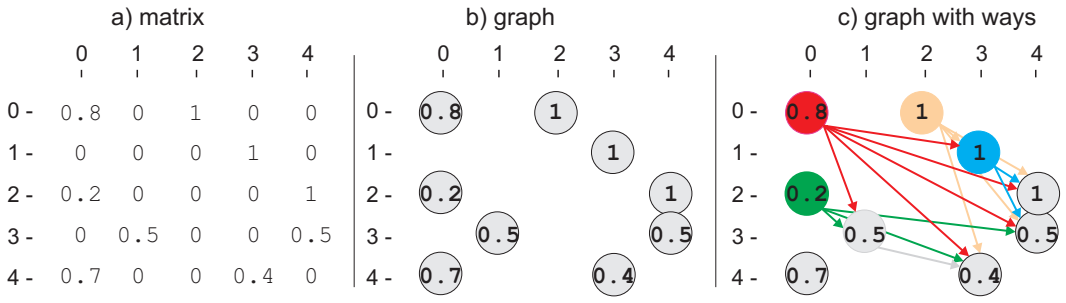


Figure 15: Example of similarity matrix and derived graph

Finding the most similar common sequence of elements ("bestWay") itself is described in algorithm 5 and commented here:

Algorithm uses nodesList as list of all nodes that is private in the class. So all functions can use it. Each node in nodeList has defaultly set weight to -1.

Algorithm step 1 - First node of bestWay can be situated anywhere in graph, probably it will be located somewhere at top-left position, but this is not a rule. So for simplify we adds startNode to the most top-left position in the graph. We can recursively use proposed function now.

There is the same problem with last node of bestWay, probably it will be located somewhere at bottom-right position, but this is not a rule. Adding one node (lastNode) with information that it is last node (weight=0), than to find children nodes of it and check if there is some child, is better for running time of algorithm.

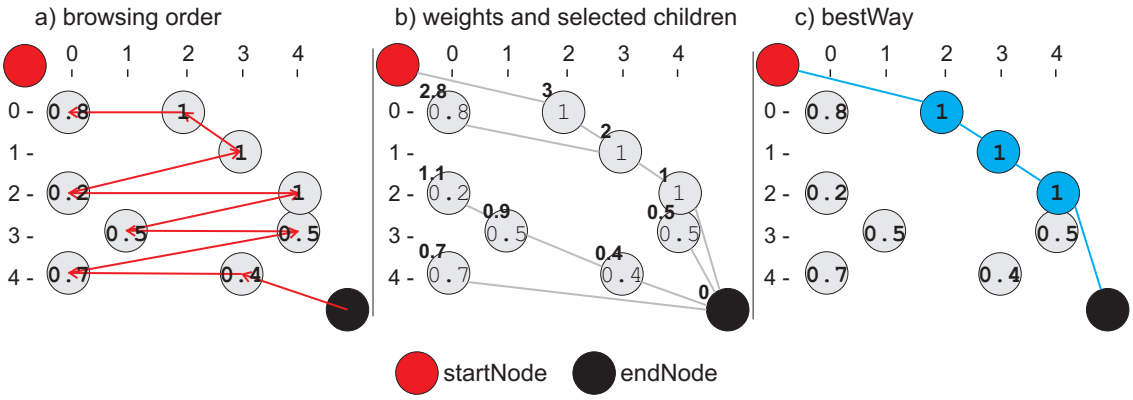


Figure 16: Example of graph browsing, weights, selected children and bestWay

Algorithm step 2 - List of orders of nodes in nodesList (nodesLinks) is created by calling function getChildrenLinks. This function selects order of each node that can be selected after given parentNode with coordinates [r,c], these children-nodes have coordinates [r+z,c+z] where $z > 0$.

Algorithm step 3 - Function that recursively computes weight of each node N starts. Weight is computed as sum of similarity of N and weight of the most weighted child of N. Additionally it sets the most weighted child as selectedChild in every node N, which is not a lastNode. Selected child and computed weights can be seen in Fig. 16 b).

Algorithm step 3 a) - Loop that goes through all children-nodes, from last node from nodeList to the previous nodes starts. This means that nodes in row (with same vertical coordinates) are browsed from the right to the left and rows are browsed from the bottom to the top. You can see this order in Figure 16 a) .

If weight of actual child is not computed (weight < 0), function calls itself with actual child as parent and with new list of nodesLinks computed for actual child.

If weight of actual child is bigger than weight of all child browsed before, its weight is set as the biggest and the link to actual child is set as selected.

Algorithm step 3 b) - If there is a selected link to child (that means that actual node is not a lastNode) it is set as selected child of actual node and weight of actual node is set to sum of actual node similarity and weight of selected child.

If there is not selected link, child of actual node is set to NULL and weight of actual node is set to 0.

Algorithm step 4 - The algorithm calls function that selects the most similar common subsequence. It only makes list from selected children nodes. Function starts from startNode and select its child, this child represents first matched couple of elements.

Algorithm 5 selecting the most similar common subsequence, LCS problem (pseudocode)

```
1.
startNode with similarity 1 and coordinates [-1,-1] is added into nodesList
endNode with similarity 1, coordinates [rn+1,cn+1] and weight 0 is added into nodesList
2.
nodesLinks = getChildrenLinks(startNode)
3.
computeWeights(startNode, nodesLinks)
4.
bestWay = getBestWay()

function getChildrenLinks(parentNode)
   FOR i < size of nodesList
      if coordinates of nodesList[i] > coordinates of parentNode
         i is ADDED to childrenList
   RETURN childrenList

function computeWeights(actualNode, children)
   highestWeight = -1
   returnedLink = -1
   bestNode = -1
3 a)
   i= (number of children) - 1
   FOR i >= 0
      IF weight of children[i] < 0
         computeWeights(children[i], getChildrenLinks(children[i]))
      if highestWeight < weight of children[i]
         highestWeight = weight of children[i]
         returnedLink = i
3 b)
   IF returnedLink IS NOT -1
      weight of actualNode = (weight of nodesList[returnedLink]) + (similarity of actualNode
          )
      selectedChild of actualNode = nodesList[returnedLink]
   ELSE
      weight of actualNode = 0
      selectedChild of actualNode = NULL

function getBestWay()
   node = selectedChild of startNode
   WHILE node IS NOT NULL
      add node into bestWay
      node = selectedChild of node
   remove last added node from bestWay
   RETURN bestWay
```

Other children are selected in the loop as child of child etc.. Last selected child is lastNode and it is removed from the list. Selected nodes are blue in Figure 16 c).

Final similarity between two rungs with number of elements rn_1 and rn_2 and with weight of first matched element w is:

$$S_R = 2 \cdot \frac{w}{rn_1 + rn_2}$$

Example of similarity computing between two rungs

Compared rungs are shown in Figure 17. Their textual representation is following:

Original rung: `XIC(T1)[XIO(T2),XIC(T3)]XIC(T4)OTE(T2)`

Updated rung: `XIC(T5)[XIO(T2),XIO(T3)XIC(T2)]XIC(T4)ADD(A,B,C)OTL(T2)`

You can see that two elements are added in updated rung. First added element is in second branch of branching and second added element is instruction ADD. Additionally tag in first instruction and last instruction in the rungs are different.

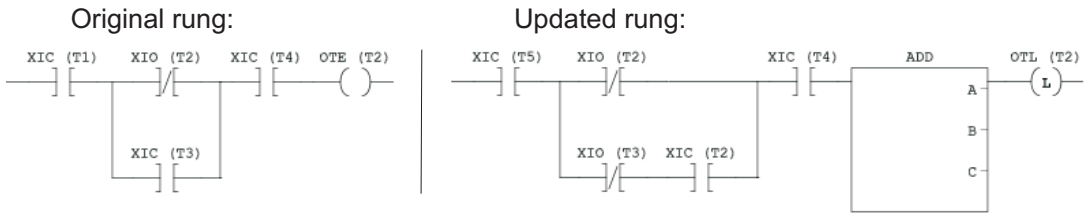


Figure 17: Example of similarity computing between two rungs

Computing similarity of branchings:

Each branch is compared with each branch from second rung and their similarities are stored in the matrix. Similarities are computed by using method for comparison of rungs. Than branches with the bigest similarity are matched as couples.

$$\begin{bmatrix} & XIO(T2) & XIO(T3)XIC(T2) \\ XIO(T2) & 2 \cdot \frac{1}{1+1} & 2 \cdot \frac{0.5}{1+2} \\ XIC(T3) & 0 & 2 \cdot \frac{0.5}{1+2} \end{bmatrix} => \begin{bmatrix} & XIO(T2) & XIO(T3)XIC(T2) \\ XIO(T2) & 1 & 0.33 \\ XIC(T3) & 0 & 0.33 \end{bmatrix}$$

$S_B = 2 \cdot \frac{1+0.33}{2+2} = 0.665$

Computing similarity of rungs:

At first each element from first rung is compared with each element from second rung and their similarities are saved as nodes of graph (see Fig. 18).

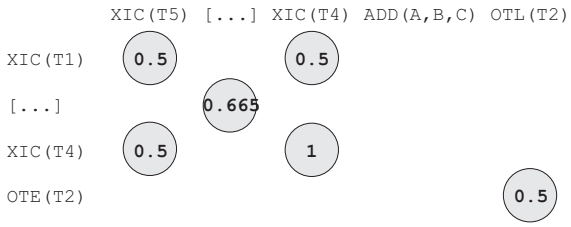
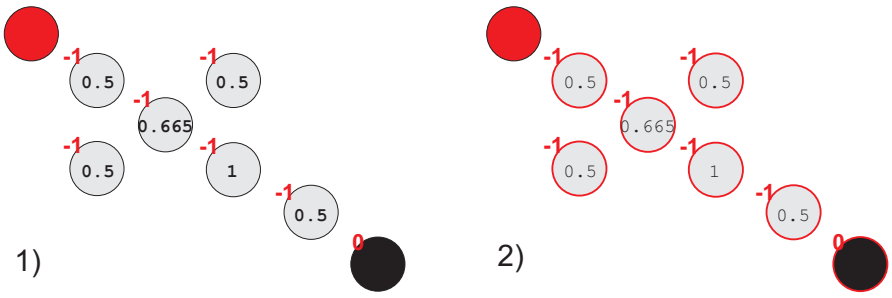


Figure 18: Concrete example of similarity computing between two rungs

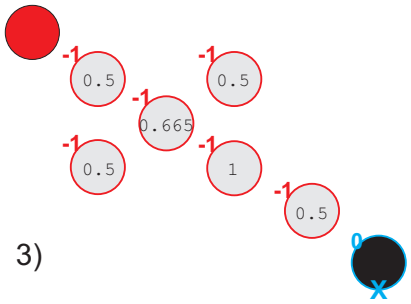
Than algorithm starts and works with given nodes only. Algorithm execution in steps:

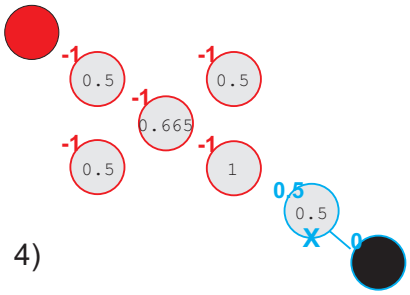
1. startNode and endNode are added to nodeList. Weights of nodes are defaultly set to -1, for endNode it is 0.
2. Children of startNode are selected.



3. Browsing of children nodes starts. It goes from bottom right to the left and upper by rows. EndNode has weight 0 and no child.

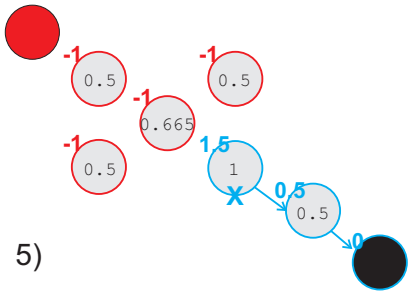
4. Next child X has only endNode between its children. So endNode is set as its child and weight of X is sum of endNode weight and X similarity.

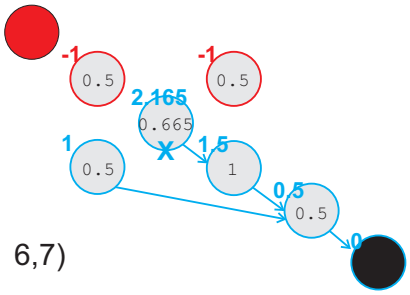




5. Next child X selects its the most weighted child between two children-nodes.

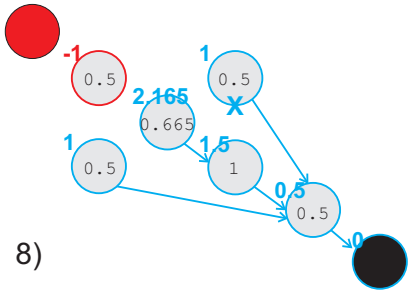
6. 7. Procedure is repeated for next children.

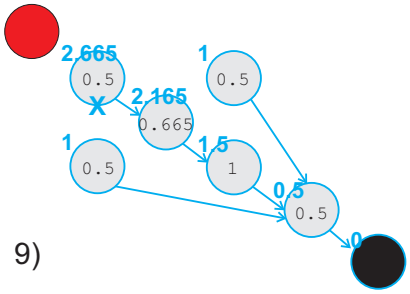




8. Note that children-nodes are minimally one line bellow and one column right from parent node.

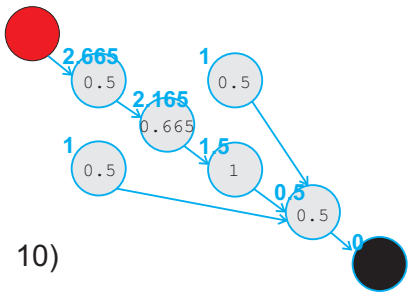
9. Weights of all children-nodes of startNode are computed.

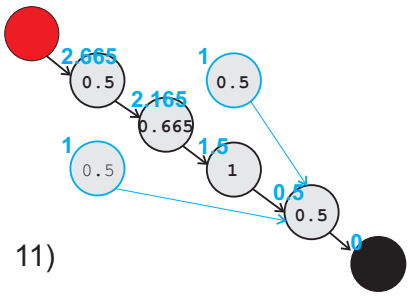




10. The most weighted child is selected for startNode.

11. Selecting bestWay starts, it selects "selected" child of startNode and then "selected" child of it etc.. Selecting ends when endNode is reached.





Similarity of rungs is then computed as:

$$S_R = 2 \cdot \frac{2.665}{4+5} = 0.592.$$

Where 2.665 is weight of first selected node, 4 is number of elements in original rung and 5 is number of elements in updated rung.

Concrete difference recognized by our algorithm (see Fig. 19):

1. First element (instruction XIC) has different tags.
2. Atomic element is added into second branch in branching XIC(T3).
3. Atomic element XIC in second branch of branching has different tags.

   Note that in second branch of branching you can not recognize if programmer has changed tag in instruction XIC and he adds instruction XIO(T3), or if he have changed instruction XIC to XIO and he adds instruction XIC(T2). This is projected in our algorithm by same similarity value between these atomic elements.
4. Instruction ADD was added into updated rung.
5. Instruction OTE was changed to OTL.

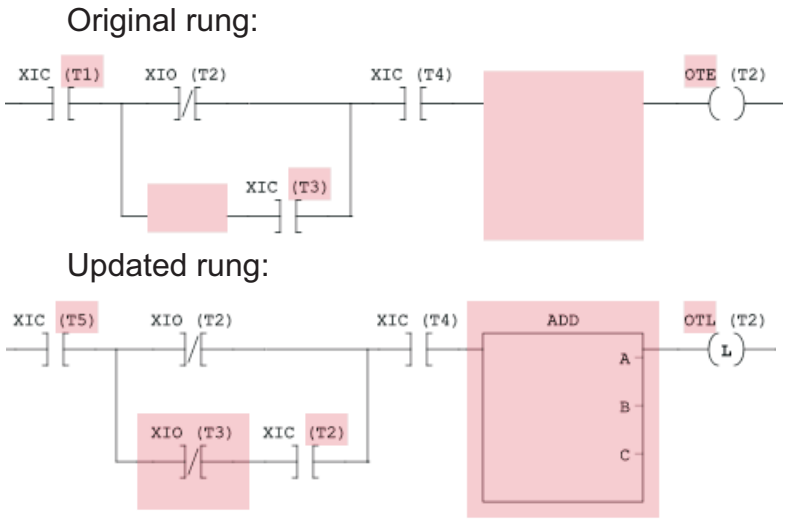


Figure 19: Specific differences recognized between rungs

4.2.4 Expressing differences

Differences in programs are divided into two parts. Differences about local tags and some other differences that was found by using method for tags comparison are stored in the first part. These differences are stored by using the same way as in Tags comparison (see 4.1.2).

Differences, which concern with program logic are stored in the second part. These differences are stored in the structure that is used in each program. Each program has list with differences in single routines. In this list one element means one routine and it stores information about names of older routine and newer routine, array of matched couples of rungs and list of rungs differences. In the list of rung differences one element means differences between one couple of rungs. This single element contains information about number of this rung in older and newer file and list of concrete differences between elements on rung (instructions, branches and branchings). Single element in the list of concrete rung differences stores information about position of difference in the rung, type of difference and concrete values.

This structure saves space in memory, because if there are two differences in same rung, they do not have to duplicate information as name of program, routine and number of rung. An

additionally it is better to work with this structure in other parts of our tool, for example while showing differences in graphical user interface.

Differences between elements

There are four types of difference between elements:

1. element is deleted in newer rung.
2. element is added into newer rung
3. instruction name is different
4. tag or tags are different

Position of difference in the rung is stored as list of integers. There are two lists of integers for each rung of couple. First number in this list means order of element in the rung. If there are more than one number in the list, it means that difference is somewhere in a branching. So numbers at odd positions in the list (list starts from zero) mean order of branch in the branching. When we are using this list we can reach position of element anywhere in the rung and the position meets our object-representation of the rung (see 4.2.3).

Example of concrete stored differences

We can see concrete recognized differences showed in ladder diagrams by our tool in Fig. 19 on the previous page. These five differences are stored as follows:

1. First instruction in the rung XIC(T1) has different tag.

   path in older rung: 0
   path in newer rung: 0
   type: 4 (tag is different)
   Concrete values:
   tag in older file: T1
   tag in newer file: T5

2. Instruction XIO(T3) was added into second branch of branching.

   path in older rung: 1/1/-
   path in newer rung: 1/1/0 (second element in the rung, second branch of branching, first element in branch)
   type: 2 (element is added)

3. Instruction XIC(T3) has different tag in second branch of branching.

   path in older rung: 1/1/0
   path in newer rung: 1/1/1
   type: 4 (tag is different)
   Concrete values:
   tag in older file: T3
   tag in newer file: T2

4. Insturction ADD was added.

   path in older rung: -
   path in newer rung: 3
   type: 2 (element is added)

5. Instruction OTE was changed to OTL.

path in older rung: 3
path in newer rung: 4
type: 3 (instruction is different)
Concrete values:

instruction in older file: OTE
instruction in newer file: OTL

4.2.5 Evaluation

Proposed method for programs comparison can find differences between local tags of programs and additionally it can recognize differences between instructions in rungs and moves of rungs. Method defaultly compares only programs and routines with the same name, but user can select what should be compared.

Method that matches rungs with no changes inside is used at first. This algorithm is finding the longest common substrings as the longest common sequences of identical rungs and it is matching couples of them until there is no free identical couple. Complexity of this algorithm is O(uv) where **u** is number of rungs in first routine and **v** in second routine. These numbers would be about one digit lesser than numbers of single instructions in routines. This algorithm recognizes moves between rungs.

Finding the most similar subsequence of elements is used on the rest of rungs, these rungs have to be parsed into the object-structure at first. Complexity of this algorithm is O(mn) where **m** is number of elements in older rung and **n** in newer rung. Because there will be probably a huge number of elements in both files, this algorithm is used only to rungs that are not matched by the previous algorithm.

Results of program comparison are list of differences between tags and list of differences between routines. Second list contains information about moves of rungs and concrete differences in all rungs. These differences are very similar to differences that human can recognize.

4.3 AddOn-Instructions comparison

Add-On Instructions are saved in L5X file in a structure that is very similar to the structure that carries information about programs. So we can use the same method as in programs comparison (see 4.2 on page 24) and we get results with the same quality.

Differences in the structure are that for example tags are not stored under element Tags/Tag as it is in programs, but they are under LocalTags/LocalTag.

4.4 Comparison of other parts of L5X file

Differences between other parts of L5X files (other than Tags, Programs and Add-On Instructions) can be found by using method that is used for tags comparison (see 4.1 on page 20). This method finds all differences in project structure, security, modules, tasks etc..

5 GUI

Proposed graphical user interface (GUI) shows recognized differences in user-friendly way. It is much faster for users to find differences in ladder diagrams if they can see it presented in same way as it is in RSLogix 5000.

GUI is divided into two main parts. In the first part, which is on the left side there is a project structure that is similar to the project structure used in RSLogix 5000. Additionally in this structure are icons signaling state of elements. The state can be *changed* (!), *deleted* in newer file (-), *added* into newer file (+) and *not-changed* (ok) (see Fig. 20). Second part is a main window, where differences are shown. There can be three things in the main window:

There are two tables for differences in tags (local and global). On the left side are tags defined in the older file and on the right side are tags defined in the newer file. Differences are highlighted by the icon and background color of table columns (see Fig. 21).

Differences in ladder diagrams are shown in three windows. On the left side is a ladder diagram from the older file. In the middle pane there is matching between rungs shown and on the right side is ladder diagram from the newer file (see Fig. 20).

Differences in other parts of L5X files are indicated in the project structure and concrete information is shown only in single window as text. These differences are presented in the same format as differences printed into command line (see Fig. 22).

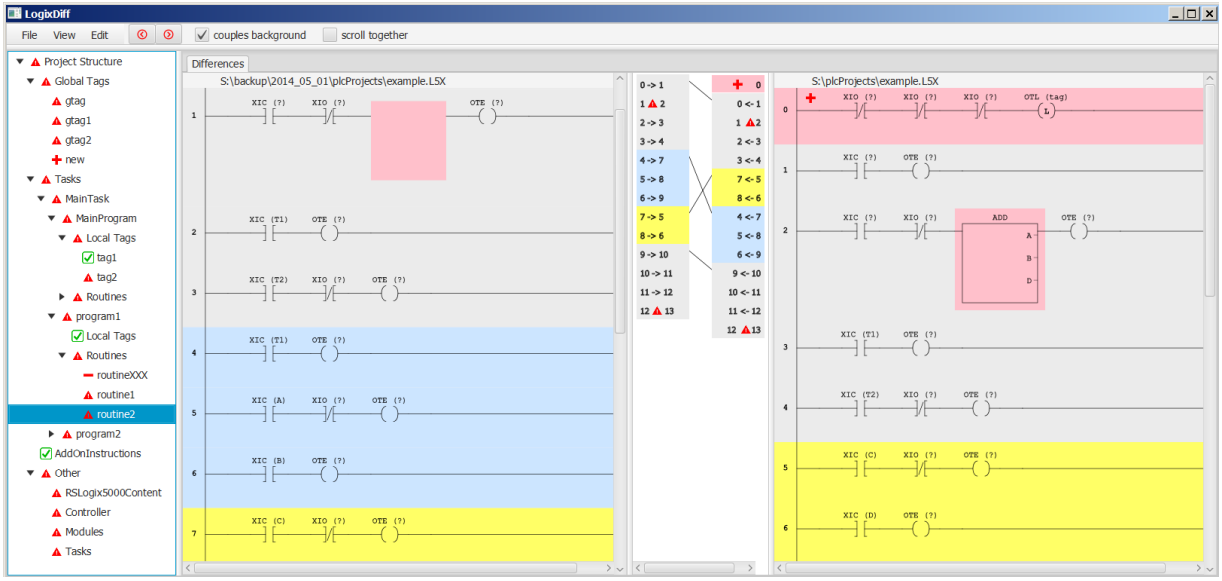


Figure 20: LogixDiff, routine differences

Each function that can be used in LogixDiff operated from command line, can be used in the GUI as well. Additionally there are options to browsing differences by clicking "next" or "previous" buttons and to show only *different* / *added* / *deleted* / *ok* elements in the project structure.

5.1 Differences in tags

The user can see all global tags or all local tags of selected program or add-on instruction by clicking on "Global Tags" / "Local Tags" in the project structure. Then tags are shown in same order on both sides and the user can see differences highlighted by orange background color. As most useful information we have selected tag name, data type and data value. These information are shown for each existing tag in the table. All other differences between tags are in the last column of tables, because changes in these information are less common. They are shown when mouse is over the cell.

The user can select only one tag. It is shown in same way as group of tags, but there is only one row in the table.

5.2 Differences in routines

Differences between routines are presented in ladder diagrams.

Middle pane where the user can see moves of rungs is between two ladder diagrams. Blocks of rungs that are moved same way have the same background color. These colors are selected from

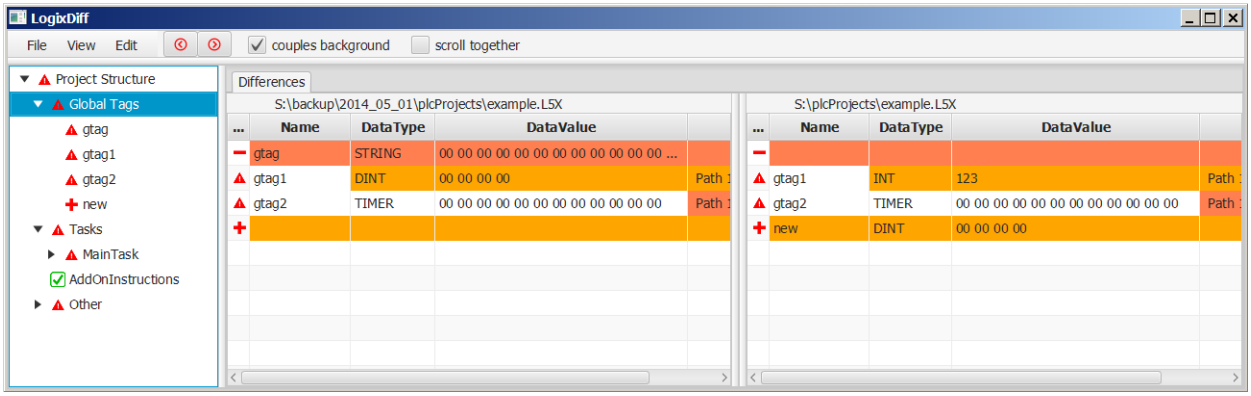


Figure 21: LogixDiff, Global tags differences

list of eight colors and it tries to use only few of them. It can happen that more blocks have same color, but algorithm checks that these blocks will not be next to each other (see Fig. 20). The user can show rungs of this block in both ladder diagrams by clicking to them.

Note, that deleted or added rungs have reserved pink background color.

In ladder diagrams is defaultly set to show same background as it is in the middle pane, this option can be deactivated, but pink color for deleted or added rungs stay there. Differences between single instructions are highlighted by pink background color as well. As you can see in Fig. 19 on page 35, instructions that are different in name have pink background under name of instruction. Instructions with different tags have pink background under changed tags. Deleted and added instructions are all on a pink background and in the second ladder diagram is free space only with pink background. Pink background is used for highlighting differences in branchings as well.

There is an option to show value that is in second routine above the difference in the first routine by using red color. It can be activated in the menu.

5.2.1 Ladder diagrams

When the user selects a routine that should be shown. Creation of object structure starts. The selected routine is found in L5X file at first. It is parsed into object structure (see 4.2.3 on page 29). Then the found differences are assigned to elements in the objected structure. This updated object structure is stored until the user selects something else in the project structure, because if the user scrolls down in routine, routine is redrawed and there is no time for finding in what element should differences be drawn.

Each instruction in object structure can draw itself. Drawing is called in a loop for each rung and rungs only passes to instructions position where they would be connected to lines. Only rungs at wanted position are drawn, that makes scrolling through huge ladder diagrams very dynamical and quick. For the user there is no difference between scrolling in routine with 15 rungs and in routine with 1000 rungs.

Position of free space in a second routine (space that highlights deleted / added elements from the first routine) is located in front of element from the second routine that is matched with the next element in first routine.

6 Case study

Tests that check runnings times of used algorithms and accuracy of results are described in this chapter.

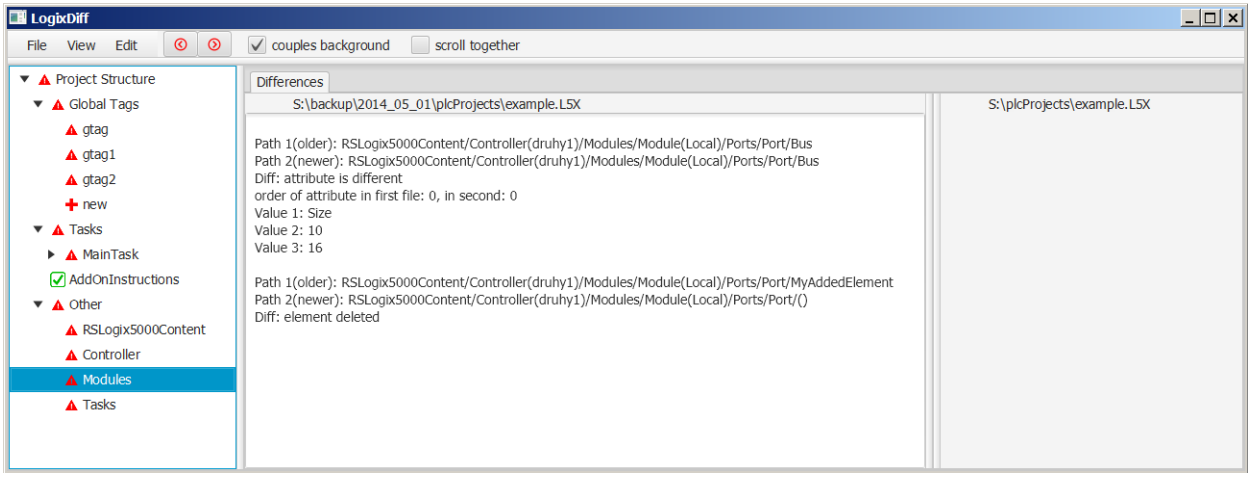


Figure 22: LogixDiff, Other differences

L5X file "TestO" with the same structure as files from RSLogix 5000 was generated as first. Size of TestO is 1044 kB and it contains 125 tags, 2 add-on instructions and 7 programs in sum of 36 routines. Routines contain 7867 rungs together, the biggest routine contains 2143 rungs. There is on avarage 6 instructions in the rung that means 47202 instructions in the whole project.

This file was copied to TestN, where we made few changes. Specifically one renamed tag, one deleted tag, changed datatype in one tag. Change of TimeZone in WallClockTime element. And about hundred of differences in rungs (deleted / moved / changed). These files are on attached CD.

Tests were operated in the LogixDiff GUI and here are avarage values from ten measurements. Values are rounded to miliseconds. Used laptop has CPU Intel Core2 Duo T5870 2x2GHz, 3GB RAM and OS Windows 7.

Running time of the algorithm that compares XML structure (see 4.1.2) and finds differences in tags was under 1ms.

Running times of the algorithm, which finds the longest common blocks of identical rungs (see 4.2.2) are in the table. The column "rungs" is number of rungs (it is same for both files), "SM" means time that costs comparison of each rung with each rung from second file and then filling the similarity matrix, "LCSS" is time that was needed for find the longest common substrings in the matrix.

rungs	SM [ms]	LCSS [ms]	sum [ms]
211	35	0.4	35.4
422	130	1.1	131.1
1055	1102	10	1112
2143	6325	17	6343

You can see that the slowest part is the strings comparison while filling the similarity matrix. Complexity of the algorithm in this part is $O(n^2)$. We can not reduce the complexity without loss of accuracy of results.

But there can be done an optimalization if the number of rungs will be commonly in thousands. And if there is some probability that rungs should not be duplicated in the routine. we can compare strings between rungs that has no couple at first and if we find couple between them, it means that we do not have to check other matched rungs, because they are different.

On the other side, if there is some probability that rungs will be duplicated. Then if an identical rung is found we can check values in same column of similarity matrix whether there is some

other matched couple at first. Then if the couple exists we can automatically match all couples that are matched between the duplicated rungs. Other possible solution is selecting of only one rung from group of duplicated rungs and then to compare only these rungs.

Second test checks the algorithm for selecting the most similar sequence of tokens (instructions and branchings in parsed rung, see chapter 4.2.3). The procedure was same as in first test, same computer, ten measurements, rounded to miliseconds. In the table are times of LCS method, column "tokens" means number of elements in the rung, "duplicities" means that there were only 5 different instructions, "branchings" says whether rung contains branchings.

For better imagination the time needed for compare only strings that contain 20 instructions was 0.02 ms and 0.05 ms for 200 instructions.

tokens	duplicities	branchings	time [ms]
20	✓	✓	0.3
20	✓		0.4
20			0.1
200	✓	✓	241
200	✓		3228
200			5

We should say that 200 instructions in series (with no branchings) in one rung is extremly huge number. Additionally duplicities between instructions in series do not make sence. That is why there is a huge time (3.228 s), because graph that is created from similarities between couples contains many nodes in this case and this slows proposed method down. We assume that there will not be many duplicities and this is the reason to use graph instead of matrix.

Results of both methods were correct and each difference was recognized.

7 Conclusions

The proposed tool LogixDiff meets all requirements of the task.

I have studied characteristics of XML files (see 3.1.1), PLC programming and the project structures of L5X files, the structure and characteristics of projects in RSLogix 5000 at first (see 2), which was very usefull in next phases of the work.

Then the most appropriate processor of XML structured files have been chosen. It is JDOM (see 3.1.2) because it provides Java representation of XML documents. It leverages SAX and DOM parsers.

The last step before the implementation was getting familiar with the existing ways of comparison of XML files and additionally with the existing ways of comparison of text files and algorithms (see 3). Only few methods were appropriate to use for the best solution. Comparison of tree-structures and other XML comparison methods are not needed. The fact we know about the project structure of L5X files is sufficient for our faster algorithm for XML structure comparison. The longest common substring method and the combination of tokens created from instructions and modified longest common subsequence method were used from known methods for text and algorithm comparison.

The implementation of comparison algorithm was divided into two parts, because there is a big difference between the structure of data that holds information about program logic and the structure of data that holds information about the project structure, tags, modules etc..

The first part of the proposed algorithm finds differences between tags. It uses only specific characteristics of XML and L5X files, which means that some XML elements have attribute *Name*. The "Name" attribute is unique on the selected position of L5X document. For example, there can not be two global tags with the same name, but there can be a program with the same

name as a tag. That is a reason to use attribute "Name" as identifier. This method is used for recognizing differences in all parts of the project, besides differences in ladder diagram (see 4.1).

The results of this method are correct and all differences are identified. Additionally the running time of this method is very fast. Complexity of the proposed algorithm is $O(n^2)$, but number of operations executed while running is much less than n^2 that is teoretically needed (see 4.1.3).

The second part complements the first part that has insufficient results in comparison of programs logic. The first part is able to recognize only if two rungs (lines of logic) are completely identical or not. The second part of algorithm works only with strings that are textual representation of rungs from ladder diagrams. It matches all unchanged couples of rungs by recursively finding the longest common substrings (see 4.2.2). We have unmatched couples of rungs now. These rungs are parsed into object-structure of java objects (tokens). Then the graph of similarities between single tokens of two rungs is created, where the most similar sequence of couples is found (the longest common subsequence method). (see 4.2.3) This is done for each unmatched rung, so we have computed similarity between each rung from the first file with each rung from the second file. The last step is matching the couples by looking for the most similar couples of rungs.

Results of this method are very similar to differences that the man can recognize. Running time of algorithm is shorter, because most of the rungs is matched without parsing and comparing every single instruction (see 4.2.5).

The algorithm gets complete information about all differences in projects. Chapter 4.1.2 describes what information about every single difference is stored for differences in project structure. Chapter 4.2.4 describes these information for differences in logic.

The proposed tool gets results for common sized files (200 kB) in reasonable time (1 s). Concrete tests are described in chapter 6. In general we can say that time depends on the size of the biggest routines (quadratic complexity of the algorithm) and on the difference level between rungs, because parsing and comparing rungs is more time-consuming than comparing two strings.

LogixDiff can operate in two ways. It defaultly starts with graphical user interface (GUI), where the structure of project can be seen. GUI can show differences in logic directly in ladder diagrams. Other differences can be opened from the project structure as well. More information about the GUI, including screenshots, can be found in chapter 5. When input parameters are passed to LogixDiff while it is starting, it starts without GUI and prints results of comparison to the command line.

The proposed tool recognizes differences in ladder diagrams much more precisely than RSLogix Compare (see chapter 2.5) and these differences are presented to the user in more clearly way. You can compare results of LogixDiff and RSLogix Compare used on the same couple of routines (see Fig. 23 and 24). You can see that differences recognized by RSLogix Compare are confusing, for example, there is only change in one tag in the second rung and it is classified by Logix Compare as two different rungs.

LogixDiff will be used by developers of Rockwell Automotion company to compare developing projects and to see differences between used Add-On Instructions. It can help to reduce duplicities between used Add-On Instructions. Additionally it can be used as a "lightweight" and fast ladder diagrams viewer. Its size is about 300 kB.

7.1 Further development

While matching the most similar non-identical couples of rungs we do not take the results of matching of identical blocks of rungs into account. If the results of matching non-identical couples were too disordered we could try to match couples of rungs that make these blocks larger at first.

Renamed and frequently used tag slows the algorithm down, because method for finding blocks of identical rungs will not match rungs with this tag. If it occurs often, we can implement an option where user selects renamed tag and it will be replaced in the document before comparison.

Morever, we can try to work on recognizing differences in the functionality of logic.

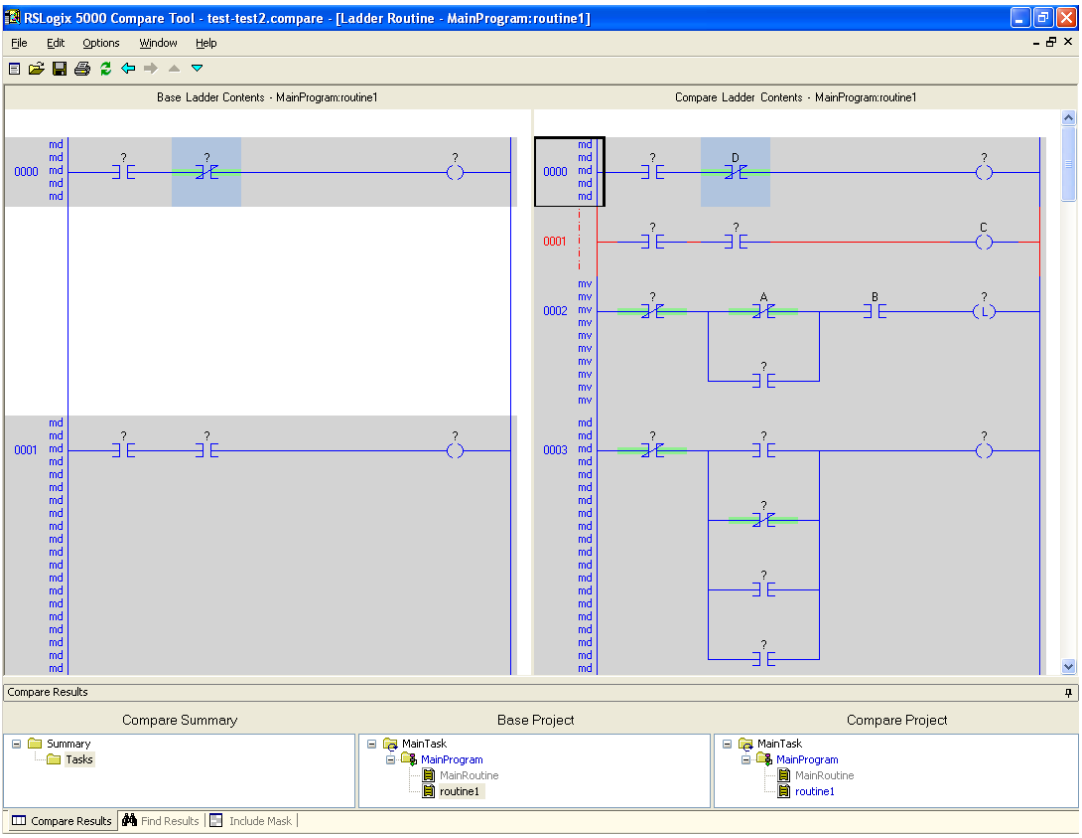


Figure 23: Example of differences recognized by RSLogix Compare

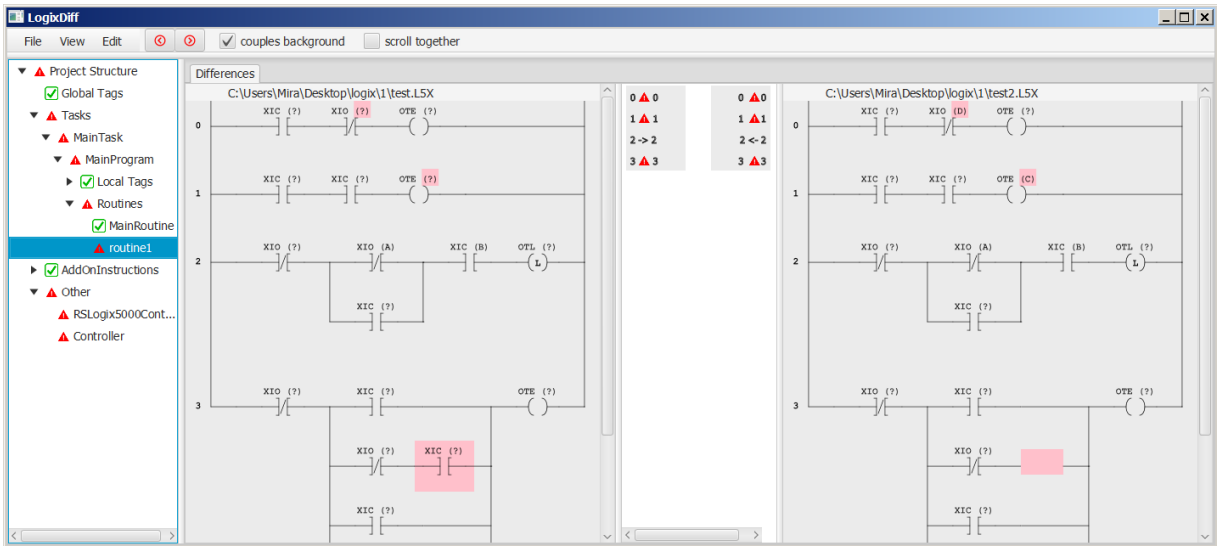


Figure 24: Example of differences recognized by LogixDiff

Abbreviations

PLC - Programmable Logic Controller

AI - Add-On Instruction

LD - Ladder Diagram

XML - Extensible Markup Language

LCS - the Longest Common Subsequence

LCSS - the Longest Common Substring

L5X file - file with .L5X extension

LogixDiff - name selected for the proposed tool

Content of included CD

```
+ source
+ examples
. thesis.pdf
. LogixDiff.jar
. readme.txt
```

Folder source contains all source codes of LogixDiff as project generated by NetBeams IDE.

Folder examples contains examples of L5X files.

References

[1] *Apache Subversion* [Online]. Available: `http://www.w3.org/TR/xpath20/`.

[2] *Concurrent Versions System* [Online]. Available: `http://www.nongnu.org/cvs/`.

[3] *Git* [Online]. Available: `http://git-scm.com/`.

[4] *WinDiff* [Online]. Available: `http://msdn.microsoft.com/en-us/library/aa242739(v=vs.60).aspx`.

[5] Rockwell Automation, *Logix5000 Controllers IEC 61131-3 Compliance Programming Manual*, (2008, July).

[6] Rockwell Automation, *Logix5000 Controllers Ladder Diagram Programming Manual*, (2012, November).

[7] Rockwell Automation, *Logix5000 Controllers Add-on Instructions Programming Manual, 1756-PM010E-EN-P*, (2012, September).

[8] *Extensible Markup Language (XML) 1.0 (Fifth Edition)* [Online]., (2008, November 26). Available: `http://www.w3.org/TR/REC-xml/`.

[9] *Simple API for XML (SAX)* [Online]. Available: `http://www.saxproject.org/`.

[10] *Document Object Model (DOM)* [Online]. Available: `http://www.w3schools.com/xml/xml_dom.asp`.

[11] *Streaming API for XML (StAX)* [Online]. Available: `http://en.wikipedia.org/wiki/StAX`.

[12] *XML Path Language (XPath) 2.0 (Second Edition)* [Online]., (2010, December 14). Available: `http://www.w3.org/TR/xpath20/`.

[13] *JDOM* [Online]., (2000). Available: `http://www.jdom.org/`.

[14] *JAXP* [Online]. Available: http://docs.oracle.com/javaee/1.3/tutorial/doc/IntroWS4.html.

[15] L. Peters, "Change detection in xml trees: a survey," 2005.

[16] K. Zhang, "A new editing based distance between unordered labeled trees," in *Combinatorial Pattern Matching* (A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, eds.), vol. 684 of *Lecture Notes in Computer Science*, pp. 254–265, Springer Berlin Heidelberg, 1993.

[17] S. Zhang, C. Dyreson, and R. Snodgrass, "Schema-less, semantics-based change detection for xml documents," in *Web Information Systems WISE 2004* (X. Zhou, S. Su, M. Papazoglou, M. Orlowska, and K. Jeffery, eds.), vol. 3306 of *Lecture Notes in Computer Science*, pp. 279–290, Springer Berlin Heidelberg, 2004.

[18] *Longest common subsequence problem* [Online]. Available: http://en.wikipedia.org/wiki/Longest_common_subsequence_problem.

[19] M. Elhadi and A. Al-Tobi, "Refinements of longest common subsequence algorithm," in *Computer Systems and Applications (AICCSA), 2010 IEEE/ACS International Conference on*, pp. 1–5, May 2010.

[20] L. Bergroth, H. Hakonen, and T. Raita, "A survey of longest common subsequence algorithms," in *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*, pp. 39–48, 2000.

[21] *diff, IEEE Std 1003.1 (2013 Edition)* [Online]. Available: http://pubs.opengroup.org/onlinepubs/9699919799/utilities/diff.html.

[22] *Longest common substring problem* [Online]. Available: http://en.wikipedia.org/wiki/Longest_common_substring.

[23] *Levenshtein distance* [Online]. Available: http://en.wikipedia.org/wiki/Levenshtein_distance.

[24] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *Software Engineering, IEEE Transactions on*, vol. 33, pp. 577–591, Sept 2007.

[25] P. A. Khaustov, "The ncp algorithm of fuzzy source code comparison," in *Strategic Technology (IFOST), 2012 7th International Forum on*, pp. 1–3, Sept 2012.

[26] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Software Maintenance, 1998. Proceedings., International Conference on*, pp. 368–377, Nov 1998.